CS 119

AD 692681

MPL

MATHEMATICAL PROGRAMMING LANGUAGE

BY

RUDOLF BAYER
JAMES H. BIGELOW
GEORGE B. DANTZIG
DAVID J. GRIES
MICHAEL B. MCGRATH
PAUL D. PINSKY
STEPHEN K. SCHUCK
CHRISTOPH WITZGALL

TECHNICAL REPORT NO. CS 119

MAY 15, 1968

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

94

MPL

MATHEMATICAL PROGRAMMING LANGUAGE

by

Rudolf Bayer
James H. Bigelow
George B. Dantzig
David J. Gries
Michael B. McGrath
Paul D. Pinsky
Stephen K. Schuck
Christoph Witzgall

Computer Science Department
Stanford University
Stanford, California

Research partially supported by National Science Foundation Grant GK-6431; Office of Naval Research Contract ONR-N-00014-67-A-0112-0011 and Contract ONR-N-00014-67-A-0112-0016; U.S. Atomic Energy Commission Contract AT[04-3] 326 PA #18; National Institutes of Health Grant GM 14789-01 A1; and U.S. Army Research Office Contract DAHC04-67-C0028.

MPL

MATHEMATICAL PROGRAMMING LANGUAGE

PART I

A SHORT INTRODUCTION

Rudolf Bayer
James H. Bigelow
George B. Dantzig
David J. Gries
Michael B. McGrath
Paul D. Pinsky
Stephen K. Schuck
Christoph Witzgall

The purpose of MPL is to provide a language for writing mathematical programming algorithms that will be easier to write, to read, and to modify than those written in currently available computer languages. It is believed that the writing, testing, and modification of codes for solving large-scale linear programs will be a less formidable undertaking once MPL becomes available. It is hoped that by the Fall of 1968, work on a compiler for MPL will be well underway.

The language proposed is standard mathematical notation. This, at least, has been the goal. Whether or not there is such a thing as a standard notation and whether or not MPL has attained it, is up to the reader to decide.

The Manual to MPL comes in three parts

FORWARD

Mathematical programming codes for solving linear programming problems in industry and government are very complex. Although the simplex algorithm (which is at the heart) might be stated in less than twenty instructions nevertheless error checks, re-inversion, product-form inverses for compactness, compacting of data, special restart procedures, sensitivity analysis, and parametric variation are necessary for practical implementation. Twenty thousand instructions are not uncommon. The cost to program such a system is several hundreds of thousands of dollars.

Recently, there has been much interest in extending mathematical programming codes into the large-scale, nonlinear, and integer programming areas. The large-scale mathematical programming applications are among the largest mathematical systems ever considered for practical solution by man. For example, a system of close to a million variables and thirty five thousand variables has already been solved using the decomposition principle.

If large-scale dynamic linear programs could be successfully solved it would have enormous potential for industrial, national, and international long-range planning.

For this reason, there is considerable interest in solving large-scale dynamic systems. Many papers have been written on this subject and the number of theoretical proposals now number in the hundreds. Very little in the way of empirical tests have been made. Occasionally, a "soft-ware" company has dared to go from a theoretical proposal to a commercial program with inclusive results. It is like going from a drawing board to a battleship when all that has been built before has been a rowboat.

The need then is to be able to write elaborate codes for solving mathematical programming systems; to test them out on sample problems; and to compare them with competitive and modified codes. Present day computer languages like FURTRAN, ALGOL, PL/1 are not in the same world as machine language of 0 1 bits. Nevertheless, it is a formidable undertaking to read codes in these languages, particularly when they involve some twenty thousand instructions. The finding of

errors (debugging) is time consuming. It is often difficult for the author of a program to decipher his own hierogliphics assuming he is available for consultation. This difficulty becomes ever more acute when extended to proposals for solving large-scale systems. It is one of the chief stumbling blocks to progress in getting practical large-scale system codes.

For this reason, the chief effort of MPL has been directed towards readability. The objective is not to invent a powerful new language but to have a highly readable language, hence one easy to read, correct, and modify.

The Iverson Language is an example of a powerful language. With a small amount of effort it could have been set up in standard mathematical notation and made readable (to a non-expert) as well. It is probably possible to implement MPL by using Iverson Language as a translator. This is not our plan.

It is possible to view MPL as nothing more than a beefed-up ALGOL or FORTRAN. The new programming language PL/1 is very powerful and could also be used to realize MPL. This is being considered. Moreover, recently there have become available excellent compilers for compilers that make easier the job of developing a compiler that would directly translate MPL into machine language. We are seriously considering this as our approach for implementing MPL.

COMPARATIVE MATH VS MPL NOTATION

The short introduction (Part I) that follows is not a formal description of the language. This is done in Part III; nor is it a general manual as Part II; rather our purpose is to motivate the need for MPL and to provide a short comparision with standard mathematical notation. MPL notation assumes that a standard key-punch or its equivalent is all that is generally available at present for program preparation. This limits the alphabet to Capital Roman and replaces $A_{i,j}$ by its functional equivalent A(I,J).

	MATH	MPL
SUBSCRIPTS:	$A_{i,j}$	A(I,J)
SUPERSCRIPT:	$A^{k}_{i,j}$	A(K)(I,J)
MATRICES:	A	A
Matrix Addition	A+B	A+B
Matrix Product	AB or A·B	A*B
Transpose	A' or A^T	TRANSPOSE(A)
Inverse	A^{-1}	INVERSE(A)
A=Matrix, k=Scalar, L=Scalar	A/K	A/K
	AK	A*K
	KA	K*A
	KL	K*L
Composing a matrix M from submatrices A, B, C, D	$M=\begin{bmatrix} A & B \\ C & D \end{bmatrix}$	M := (A,B)# (C,D); or M := (A,B)#(C,D);
	M=(A,B,C)	M := (A,B,C);
Column of a matrix A	$A_{\cdot,j}$	A(*,J)
Row of a matrix	$A_{i,\cdot}$	A(I,*)
Determinant	$\|A\|$	DETERMINANT(A)
Array of Consecutive Integers	(k, k+1,...,ℓ)	(K,...,L)

	MATH	MPL
OPERATORS:		
Matrices or Scalars:		
Addition, Subtraction, Multiplication	+, -, ·	+, -, *
Division by Scalar	A/k	A/K
Exponent	A^2	A**2
	A^{-1}	INVERSE(A)
Sign	2, +2, -2	2, +2, -2
Substitution Operator(=)	New value of A=value of B+C	A := B+C; (meaning: change the value of A on LHS to equal the value of B+C on RHS.)
Logical Operators	AND, OR, NOT	AND, OR, NOT

MATH: If A > B, C $\geq$ D, and not D = 0
MPL: IF A > B AND C >= D AND NOT D = 0 THEN

MATH: If A > B or C > D,
MPL: IF A > B OR C > D THEN

	MATH	MPL
Relational Operators	=, <, >, $\geq$, $\leq$, $\neq$,	=, <, >, >=, <=, ¬=,
Set Operators	A $\cup$ B, A+B	A OR B
	A $\cap$ B, A·B	A AND B
	A $\cap$ (not B), or A $\cap \bar{B}$	A AND NOT B
MAPPINGS, PROCEDURES, SUBROUTINES:		
B, X, Y...Matrices, Sets, Scalars	Y = F(X)	Y := F(X);
	Y = SIN(X)	Y := SIN(X);
	$Y = 2BX^2$	Y := 2*B*(X**2);
	$Y = B^{-1}$	Y := INVERSE(B);

	MATH	MPL
SYMBOL REPLACEMENT:	Let $W = f(x,y)$	LET W := F(X,Y); (meaning do not compute W but replace it by F(X,Y) wherever W appears later on.)
SETS:	(any set of elements)	(Index sets only)
	$S = \{1,3,-2,5\}$	S := SET(1,3,-2,5);
	$S = \{1,\ldots,n\}$	S := (1,...,N);
	$I \in S$	I IN S
	$I \in A \cup B \cup C$	I IN A OR B OR C
	$I \in A \cap B \cap C$	I IN A AND B OR C
	$I \quad A \cap \bar{B}$	I IN A AND NOT B
	$D = (A \cup B) \cap C.$	D :=(A OR B) AND C;
Index Set or Domain of a vector A	Domain of A	DOM(A)
Index Set of a matrix A	Row Domain of A	ROW_DOM(A)
Defining of Set where P(I) a Boolean Expression or property is true	$\{i \in R : P(I) = \text{true}\}$	(I IN R|P(I) = TRUE)
or	$\{I \in R : P(I)\}$	or (I IN R|P(I))
or	$\{i \in R \| P(I)\}$	
	$\{i \in R \| A_i > 0\}$	(I IN R|A(I) > 0)
	$\{i \| A_i > 0\}$	(I IN DOM(A)|A(I) > 0)
Empty Set	$\emptyset$, Null, Empty	NULL
SET FUNCTIONS:		

Suppose $S = (S_1,\ldots,S_m)$ is a 1-dimensional array of integers and we wish to pick out column vectors $A_{s_1}, A_{s_2}, \ldots, A_{s_m}$ to form a matrix B.	$B = (A_{s_1}, A_{s_2}, \ldots, A_{s_m})$	B := A(S);
		or B := (A(J) FOR J IN S);
		or B := (A(S(I)) FOR I IN (1,...,M));
		However,
		B := (A(S(1)),..., A(S(M))) is not correct because (P,...,Q) means (P,P+1,P+2,...,Q) in MPL

	MATH	MPL
SYMBOLS:		
CAPS	A, B, ---	A, B, ---
Lower Case	a, b, ---	(not available yet)
Greek	α, β, ---	(not available yet)
Integers	0, 1,...,99, ---	0, 1,...,99, ---
Multi-Character Symbol:		
as function name:	PIVOT(M,R,S) SIN(X)	PIVOT(M,R,S) SIN(X)
as variable name:	(not used)	B2, BASIS, X_S
Brackets	{} []	(not available yet)

SYNTAX

In general, a procedure has the form:

```
PROCEDURE  F(X,Y,Z)
Statement;
   ---
Statement;
FINI;
```

Certain reserve words like FOR and IN can be interspersed in place of commas in F(X,Y,Z) as in the example given below.

Example Given an array of integers R, we wish to write an algorithm, called SUM, that yields $S = \sum_{j \in R} F(j)$.

```
    PROCEDURE  SUM(F)
                                       "SET UP A STORAGE REGISTER  S  TO ACCUMULATE
                                       THE SUM OF TERMS.  INITIALLY,"
(1):    S := 0;                        "LET  S'  BE THE UPDATED VALUE OF  S.  WE WANT
                                       TO STORE  S'  IN THE SAME PLACE AS  S  AND
                                       THEREAFTER CALL IT  S."
(2):    SAME LOCATION(S,S');
(3):    S' := S + F(I) FOR I IN DOM(F); "ITERATIVELY ADDS  F(I)  TO  S."
(4):    SUM := S;                      "SETS THE VALUE OF THE FUNCTION EQUAL TO S"
(5):    RETURN;                        " 'RETURN' MEANS:  RETURN TO MAIN ROUTINE."
        FINI;                          " 'FINI'  MEANS:  END OF WRITE-UP."
```

Once the $\sum$ symbol, or rather SUM, is in the procedure library we can use it to write a statement like $P = \sum_{1}^{n} i^2$ in MPL.

```
P := SUM(I**2  FOR  I  IN  T)    WHERE  T := (1,...,N);
```

The reference numbers like (1), (2),..., on the left are called labels. They are not necessary in the above example and may be omitted. Labels can be a string of characters or numbers like (1), (2). If the latter, they need not be consecutive. Labels are used to locate a statement when a program branches.

A statement like the one with label (3) is called a substitution statement because S' := S + F(I); means: Substitute for the current value of S' on the left a new value equal to the current value of S + F(I) on the right.

In general, A := B; means updated A = Current B. A statement S := S + F(I); looks like nonsense but means: Updated S = Current (S + F(I)). Hence a programmer not interested in readability would probably boil down the procedure SUM to two lines.

```
PROCEDURE SUM(F)
 SUM := 0; SUM := SUM + F(J) FOR J IN DOM(F); RETURN;  FINI;
```

There are several different types of statements that one can draw upon to write a procedure:

Procedure Name	If	Define
Substitution	For	Release
Let	Same Location	Fini
Return	Go to	

and some words like "then", "otherwise", "endif", "do", endfor" that indicate different parts of a compound "if" or "for" statement.

Procedure Name Statement: PROCEDURE F(X) PROCEDURE F("IN" X, "OUT" Y) where X, Y represents a list of one or more symbols.

Examples:

```
PROCEDURE SIN(X)
PROCEDURE PIVOT(A,R,S)
PROCEDURE SIMPLEX(A,B,C,BV)
PROCEDURE ARGMIN(F(I) FOR I IN T)
```

"where ARGMIN finds the first index or argument where the minumum occurs."

Substitution Statement: A := Arithmetic Expression;

Examples:

```
S := 0; M := ARGMIN(H(J) FOR J IN R);
A := PIVOT(A,R,S); G := INVERSE(MATRIX) + H;
S := ARGMIN(C(J) FOR J IN T) WHERE T := (1,...,N);
```

Let Statement: LET A := Arithmetic Expression;

Examples:

```
LET A := B;
LET T := (I IN DOM(B)|A(I,S) > 0);
LET R := ARGMIN(B(I)/A(I,S) FOR I IN T);
```

If LET is used to simplify only one statement, a WHERE can be used instead using inverse order.

G := INVERSE(B) WHERE B := TRANSPOSE(A);

Return Statement: RETURN;

If this statement is reached during execution of the subroutine, the next step is to return to the main routine.

If Statement:

```
IF P THEN statement ;...; statement;
OTHERWISE statement ;...; statement;
ENDIF;
```

Example:

```
IF R = NULL THEN GO TO (21); OTHERWISE
A := PIVOT(A,R,S); ENDIF;
```

All statements up to "OTHERWISE" are executed if proposition p is true and then sequence control skips to the statement following ENDIF. However, as in the above example, there is a GO TO statement preceding the OTHERWISE then control skips to wherever GO TO directs. If p is not true, control skips to statements following "OTHERWISE". For the case of several parallel conditional statements OR IFstatements are available - see Part II and III. OTHERWISE can be omitted if immediately followed by ENDIF.

For Statement:

```
FOR I IN T DO statement ;...; statement; ENDFOR;
```

Example:

```
FOR I IN (1,...,M) DO
S' := S + F(I);
T' := S' + G(I);
ENDFOR;
```

<u>Same Location Statement</u>: SAME LOCATION(A,B);

A and B will be assigned the same set of storage locations in the computer. An alternative way to accomplish the same thing would be to write: LET A := B; For psychological reasons, it seems best to separate the concept: "A is another symbol for B" from t e concept "same storage location".

<u>Go to Statement</u>:

GO TO ℓ (where ℓ is a label). This means that control is to skip to the statement that has ℓ as a label.

<u>Define Statement</u>:

Example: DEFINE B DIAGONAL M BY M;

Used to define the size of storage array needed for a symbol whose value will be computed piecemeal later on.

<u>Release Statement</u>:

To release a symbol and its storage assignment a release statement takes the form:

RELEASL A,B;

Its purpose is to conserve storage and permit re-use of the same symbol for some other purpose. A special type of automatic release is available that allows release of all symbols in a block of code.

Release occurs automatically when a procedure returns to a main routine; all symbols defined in the procedure and their storage are released <u>except</u> the output symbols, which are treated as part of the symbols of the main routine.

Symbols used as dummies as G in the statement: Z := A+G WHERE G := INVERSE(M);

are treated as local to the statement and are immediately released. The same applies to the running index in a compound For statement and to a dummy parameter in a Let statement as I in : LET G(I):= B(I)/A(I,J); .

EXAMPLE: SIMPLEX ALGORITHM

```
 PROCEDURE SIMPLEX ("IN" A,B,C,BV, "OUT" BV', B', Z', CASE);

"WARNING:  ALL INPUTS ARE MODIFIED IN THE COURSE OF CALCULATIONS."

"THE PROBLEM IS TO FIND  MIN Z,  X ≥ 0  SUCH THAT:

                               AX = B,   CX = Z .

IT IS ASSUMED THAT:

                A            IS IN CANONICAL FORM WITH RESPECT TO
                BV           THE INITIAL SET OF BASIC VARIABLES.
                B ≥ 0        ARE THE  X  VALUES OF  BV, I.E.  X(BV) = B.
                             THIS INITIAL BASIC SOLUTION IS REQUIRED TO BE FEASIBLE,
                             I.E.  B ≥ 0.
                BV'          IS THE OPTIMAL SET OF BASIC VARIABLES.
                B'           ARE THE  X  VALUES OF  BV', I.E.  X(BV') = B'.
                Z'  =        MIN Z
                CASE  =      FINITE OR UNBOUNDED.
                BV', B', Z'  REFER TO LAST BASIC SOLUTION IN THE CASE THAT
                             'CASE = UNBOUNDED'."

"INITIALIZATION"

     DEFINE CASE CHARACTER;

(1):  Z := 0;                "PRIMES WILL BE USED FOR UPDATED VALUES OF VARIOUS SYMBOLS.
                             THESE WILL BE STORED IN THE SAME LOCATION."
(2):  SAME LOCATION (A, A'), (B, B'), (C, C'), (BV, BV'), (X, X'), (Z, Z');
                             "ITERATIVE LOOP"
                             "LET  S  BE COLUMN COMING INTO BASIS."
```

```
(3):  MIN_1("IN" C, "OUT" S, C_S);

                          "MIN_1 IS A FUNCTION THAT RETUNS THE INDEX AND THE
                          MINIMUM COMPONENT OF A VECTOR.  IN THIS CASE  VECTOR = C."
                          "WE NOW TEST WHETHER  X(BV) = B IS OPTIMAL."

(4):  IF  C_S = 0 THEN CASE := 'FINITE' ;  RETURN;  OTHERWISE
                          "LET  R  BE THE INDEX OF THE BASIC VARIABLE DROPPING."

(5):  MIN_1("IN" (B(I)/A(I,S) FOR I IN DOM(B) | A(I,S) > 0), "OUT" R,Q);

                          "IF ABOVE SET EMPTY, MIN_1 RETURNS  R = NULL, Q = 0:
                          OTHERWISE THE INDEX  R  AND THE MINIMUM RATIO, CALLED
                          Q, IS RETURNED."

(6):  IF  R = NULL THEN  CASE :=  'UNBOUNDED' ; RETURN;  ENDIF;

                          "UPDATE EVERYTHING BY PIVOTING ON  A(R,S), PRIMES WILL
                          BE USED FOR UPDATED SYMBOLS.  THESE ARE STORED IN SAME
                          LOCATION, SEE (2)."

(7):  B'(R) := Q;

(8):  A'(R,*) := A(R,*)/A(R,S);

                          "ROW_DOM(B) IS THE DOMAIN OF INDICES FOR B."
(9)   FOR  I  IN ROW_DOM(B) | I ¬= R  DO

(10): B'(I) := B(I) - A(I,S) * Q;

(11): A'(I,*) := A(I,*) - A(I,S) * A'(R,*); ENDFOR;

(12): C' := C - C(S) * A'(R,*);

(13): Z' := Z + C(S) * Q ;
```

(14): BV'(R) := S;

"THE REMAINING COMPONENTS OF BV ARE UNCHANGED AND SINCE BV AND BV' ARE STORED IN THE SAME LOCATION. UPDATING IS COMPLETE, RECYCLE."

(15): GO TO (3); FINI;

MPL

MATHEMATICAL PROGRAMMING LANGUAGE

PART II

GENERAL DESCRIPTION

March - 1968

Prepared by Paul D. Pinsky

Rudolf Bayer

James H. Bigelow

George B. Dantzig

David J. Gries

Michael B. McGrath

Stephen K. Schuck

Christoph Witzgall

The Manual to MPL comes in three parts

PART I: A SHORT INTRODUCTION

PART II: GENERAL DESCRIPTION

PART III: FORMAL DEFINITION

ABSTRACT

The objective is to develop a readable language for writing experimental codes to solve large-scale mathematical programming systems. Readability is defined as standard mathematical notation with minor adjustments reflecting current limitations of input-output equipment. Thus symbols are restricted to those found on a standard keypunch; subscripts (or superscripts) like $A_{i,j}$ appear as A(I,J). Starting in the Spring of 1967, several test algorithms written in the proposed language gave evidence that readability was an achievable objective.

A task group in the latter part of 1967 began to define the proposed language in BACKUS Normal Form with the intent of using a special compiler's compiler to implement the language.

TABLE OF CONTENTS

1.0 INTRODUCTION

This paper describes recent work on a computer programming language for the implementation of mathematical programming algorithms on a digital computer the objectives of the language are:

a) to facilitate programming an algorithm from theoretical form to computer code in as short a time as possible, and

b) to enable other mathematical programmers to understand and modify an existing code with a minimum of effort. The present efforts are being directed toward the coding of experimental mathematical programming algorithms rather than commercial techniques. By and large, the first report (Mathematical Programming Language, June 1967) represented the thinking of persons with mathematical programming backgrounds. Since then, several computer scientists contributing to the project have brought the language much closer to implementation.

The purpose of this report is to explain the use and the reasons for the concepts being developed in MPL. This part of the Manual attempts to explain the reasons for using the specific concepts of MPL while the third part developed under the guidance of David Gries gives a formal definition of the language in a modified form of BACKUS Normal Form. Part III is primarily the work of Stephen Schuck, who, since joining the project last summer, has been a driving force behind the implementation of MPL. His work in turn uses several concepts developed by Rudolf Bayer and Christoph Witzgall of the Boeing Scientific Research Laboratories. At present, the BACKUS Normal Form is used to describe the legal programs, not the phrase structure of the language.

David Gries of Stanford University is currently developing a technique of writing compilers, called the Kompiler Implementation System (KIS), which, it is planned will be used in the implementation of the Language. Many of the concepts

presented herein, are the same as or similar to those found in existing compiler languages (ALGOL, FORTRAN, COBOL, PL/1, etc.). One of the difficulties encountered thus far in writing a formal definition of MPL is that mathematical notation depends upon the context for its meaning. $(P_1,\ldots,P_M)$ may mean $(P_1, P_2, P_3,\ldots,P_M)$ or it may mean $(P_1, P_1 + 1, P_1 + 2,\ldots,P_M)$. This is defined in MPL to mean the latter.

There are certain concepts planned for MPL that have not yet been set down in BACKUS Normal Form. In particular, the representation of index sets has not been completely formalized; the ability to operate with matrices whose elements are matrices (useful for example in the decomposition principle) has not yet been fully developed. Procedure parameters need more work. Input-output statements have not yet been defined, nor storage commands that would reflect the variable size and speed of different memory locations.

2.0 MPL LANGUAGE ELEMENTS

The set of characters upon which MPL is built is the character set found on standard key-punches (such as the IBM 029 key-punch). For convenience, we shall group these characters into the categories of letters, digits, and special characters. The letters are A through Z, the digits are 0 through 9, and the special characters are as follows:

() < > , . + - * / : ; ¬ = " ' _ # @ % & |

and a blank. Elements of MPL are defined to be one of the following four constructs-variable, constant, operator, or reserved word. Let us now delve more deeply into each of the above elements.

2.1 VARIABLES

Variables are symbols which represent those data values which may change during the execution of the program. There are several types of variables - arithmetic, logical, set and character.

For example, if C is a row vector and Q a scalar both previously defined then

D := (C, SIN(Q));

sets up a new row vector D with one more component than C. The function sin(x) is a reserve word and "sin" cannot be used as symbol for a variable on the left hand side of an equation.

A variable may have zero, one, or two dimensions. A zero-dimensional variable is a scalar, a one-dimensional variable a vector, a two-dimensional variable a matrix.

In the remainder of this report, an array refers to any variable whose dimension is greater than zero. Each matrix has associated with it a structure shape commonly used in mathematical programming algorithms. These shapes are rectangular, diagonal, upper triangular, lower triangular, and sparce (meaning few non-zero elements). The concept of structure shape is useful in conserving memory space and execution time. An example of the use of shape matrices is in the storage and multiplication of two diagonal matrices of size $n \times n$. Storing them as diagonal in the computer requires only n memory words for each (as opposed to n^2 for a rectangular matrix), and the multiplication of two diagonal matrices requires only n elementary multiplications as opposed to n^3 for rectangular matrices. Vectors have the shape of row or column; this distinction is required for multiplying vectors by vectors or matrices. An additional feature of MPL is that the elements of an array may be arrays. This construct is helpful in coding algorithms such as the decomposition principle. Another variable allowed is an index set variable. This consists of an ordered set of integers. Examples of index sets are:

```
(1,...,M)
SET(1, 3, -4, 3, 12)
(I IN (1,...,M)|A(I,S) > 0)
```

More will be said about how to define and use variables later on.

The symbols which constitute variables have two parts, the variable name and an optional subscript. The variable name alone completely identifies the variable under consideration if that variable is a scalar or an entire storage structure (vector, matrix, etc.). If the variable represents a subset (element, row, column, etc.) of a larger array, the variable-name part only identifies the larger array, subscripts being needed to specify the particular subset. Variable names always begin with a letter, but the characters which follow it may be any number of letters,

digits, or underscores. Reserved words (defined in Section 2.4) may not be used as variable names.

Examples of variable names are

A
OBJECTIVE_1
KEY_SET
BASIS_INVERSE

However, variable names with blanks like KEY SET are not allowed. Subscripts are either scalar arithmetic expressions or the symbol * . Scalar arithmetic expressions (defined in Section 3.2) are automatically bounded to the nearest integer value when used as a subscript. The subscript * refers to an entire dimension of a storage structure. Thus

A(*, J) refers to the J^{th} column while
A(I, *) refers to the I^{th} row of the matrix A .

The following examples illustrate the use of subscripts:

M(B + C, 3)
B_INVERSE(1, *)
X_VALUE(BASIS_LIST(I)).

2.2 CONSTANTS

Constants are of four types--arithmetic, logical, set and character. The type of a constant determines how the number will be stored in the machine and used in calculations.

ARITHMETIC CONSTANTS may be either integer or real.

INTEGER ARITHMETIC CONSTANTS are written as a string of digits without a decimal point, examples 1, 10, 10090.

REAL ARITHMETIC CONSTANTS may or may not have an exponent. An exponentless real number is a sequence of digits containing a decimal point. Examples: 1., 1.0, .3925, 102.34. The exponent form of the real constant allows writing the constant in modified scientific notation. This form consists of an exponentless real number followed by an E (meaning 10 to the power) followed by an optionally signed string of digits.

Examples:

2.5E02	$(25.\times10^{2} = 2500.)$
1.0E-02	$(1.0\times10^{-2} = .01)$
.8E03	$(.8\times10^{3} = 800.)$
9.1E+05	$(9.1\times10^{5} = 910000.)$

LOGICAL CONSTANTS are TRUE and FALSE.

A SET CONSTANT is NULL.

CHARACTER CONSTANTS are any string of characters enclosed by single quotes (')

Examples:

'TABLEAU'

'PRICES ARE'

2.3 OPERATORS

Operators are the connecting elements which allow the grouping of variables and constants into larger language phrases called expressions. Operators are of five classes:

a) arithmetic operators-unary: + and - ;
and binary: + (addition), - (subtraction),
* (multiplication), / (division), and
** (exponentation).

b) logical operators-unary: NOT ;
and binary: AND, OR.

c) relational operators - = (equal),
¬= (not equal), >= (greater than or
equal), <= (less than or equal),
> (strictly greater than),
< (strictly less than).

d) concatenation operators (for building
up matrices from elements) : a
comma (,) is used for horizontal
concatenation; a number sign (#) is
used for vertical concatenation.

e) set operators - OR (union),
AND (intersection), AND NOT (relative
complement).

The use and meaning of the first three operators is quite similar to operators in existing languages (ALGOL) while the concatenation operator may be new to the

reader. This operator is used to build larger storage structures from smaller ones. For now an example of concatenation operators will be given; the detailed explanation of their use being presented in Section 3.2.3. Suppose A, B, C, and D are matrices of the same dimensions. Then M := (A, B)#(C, D); represents a larger matrix of the following form: $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$. If the programmer writes M := (A, B)#(C, D); partly on one punch card and partly on the next it takes the form

```
M := (A, B)#
     (C, D);
```

To resolve ambiguities which can develop in forming combinations of elements, each operator has an associated precedence. In the absence of parenthesis to dictate the meanings of such combinations, the meaning will be given by the precedence of the operators, with those having higher precedence being first. Operators of equal precedence will be performed from left to right as one would expect. Section 2.5.2 in Part III interprets the operator symbols in order of decreasing precedence. A * before an operator indicates that its precedence is the same as the preceding operator. The following examples show the meaning of precedence.

A - B/C + D	is interpreted as	A - (B/C) + D
(A, B)#C	is interpreted as	$\begin{bmatrix} (A, B) \\ C \end{bmatrix}$
B + C/D ** E*A	is interpreted as	B + ((C/(D**E))*A)

Ambiguous notation in two of the examples can be avoided, of course, by use of parentheses.

2.4 Reserved Words

Reserved words in MPL fall into the categories of keyword symbols or standard function names such as sin(x) and procedure names. Recall that reserved words may not be used as variable names. Keyword symbols (such as FOR, IN, END, GO TO) will be discussed in Section 4.2.3.

Functions:

A standard function name identifies a standard function. It is hoped that extensive use of standard functions will lead to ease in programming and enhance the readability of the resulting codes. Presented in Section 5, Part III is a list of standard functions, which hopefully will grow as MPL developes. Reference to a standard function is of the form V := F(P) where V represents the value of the function, F represents the name of the function, and P represents one or more arguments which we will refer to as a parameter list. Depending upon the function, the value may be integer or real, scalar, vector, or matrix, and if matrix, it may have any structure shape. These properties as well as the properties of the parameter list are described in Part III. Following are a few examples of the use of standard functions. Let C and X be vectors, A a matrix, and T an index set all previously defined:

```
Z := SUM(C(I)*X(I)  FOR I IN T);
R := ARGMIN(B(I)/A(I,S) FOR I IN T|P(I,S) > 0);
```

2.5 Comment Statements (Quote Symbols)

In the algorithms coded thus far by the MPL group, it has been found that comments are essential for readability of computer codes. Comments may be placed between any two sentences and are separated from the program by quote

marks before and after the comment. Example:

```
SAME LOCATION (COUNT, COUNT');
A := B + C;
                    "A  IS THE SUM OF  B  AND  C"
FOR  I IN SET_1, COUNT' := COUNT + 1;
                    "WHERE COUNT' IS THE UPDATED VALUE OF
                    COUNT WHICH IS STORED IN THE SAME
                    LOCATION AS COUNT AND REFERRED TO HERE-
                    AFTER AS COUNT."
```

The general objective of MPL is readability. It is however, doubtful that a program will be readable unless liberally interlaced with comments statements whereby the programmer explains to the reader why he is doing the various steps. In experiments with mathematical programming routines, almost two lines of comments are needed on the average to explain an executable line of code. Comment statements can consist of one or several lines set off at the beginning and end by quote makes.

```
                    "PIVOTING WILL BE DONE ON THE FULL
                    MATRIX  D  WHICH INCLUDES  A, THE
                    RHS  B, AND COSTS  C."

D := (A, B)#
     (C, 0);

                    "WE NOW INCREMENT COUNT AND RECYCLE."

COUNT' := COUNT + 1;  GO TO (21);
```

3.0 Expressions

Variables, constants, and operators are combined into larger language phrases called expressions. Expressions are either arithmetic, logical, set or character. In addition, the value of an arithmetic expression has a shape (rectangular, diagonal, lower triangular, upper triangular, sparce). The following sections explain the use and meaning of some of the special features of MPL expressions.

3.1 Logical Expressions

A logical expression, having the value of TRUE or FALSE, is a comparison between two arithmetic expressions. Two arithmetic expressions which are compared by a relational operator must be identical in type, form and shape. Following are examples of logical expressions:

$$A \geq B$$

$$\text{NOT } (X(I) \geq Y(I))$$

$$(Z \geq M) \text{ AND } (B + C \leq A + D)$$

$$(H(I) = Z(I)) \text{ OR } (M = Q)$$

When A and B are scalars and ρ is a relational operator, then the interpretation of $A \rho B$ is clear. However, in the case of arrays, the meaning of $A \geq B$ can differ by author. Table 1 below defines precisely what is meant by the relational operators in MPL.

TABLE 1

In this Table, A and B are arrays identical in type, form, and shape. A_i, B_i refer to elements of A and B.

MPL Statement	Mathematical Meaning
A = B	$A_i = B_i \quad \forall_i$
A ≤ B	$A_i \leq B_i \quad \forall_i$
A < B	$A_i < B_i \quad \forall_i$
A ≥ B	$A_i \geq B_i \quad \forall_i$
A > B	$A_i > B_i \quad \forall_i$
A ¬=B	$A_i \neq B_i$ for some i

3.2 Arithmetic Expressions

Arithmetic expressions are any combination of the following types--computational expressions, function references, and array builders.

3.2.1 Computational Expressions

Computation expressions are of the structure 'left-operand'-'operator' 'right-operand'. If the left operand is missing, the operator is unary (one operand) - Example: -A, + (Q-Z/B). If both operands are present, they are connected by a binary operator (two operands) - Example: A+B, C**D . At execution time the expression will be evaluated to produce a result. In addition to being defined, an operation can only be performed if the operands conform to the conventional restrictions of matrix algebra (for example - M and N are matrices, then M*N has meaning if and only if the number of columns of M equals the number of rows in N). Section 2.5 of Part III describes these relationships in detail.

3.2.2 Function References

A function reference expression involves the use of predefined functions as set forth in Section 2.4. Examples of function references used with computational expressions to form new arithmetic expressions are given below.

X*SUM(Y)

A*TRANSPOSE(B)

BASIC_COSTS*INVERSE(BASIS)

We shall see further use of function references in array builders in the next section.

3.2.3 Array Builders

There are two types of array builders--concatenators and array designators.

A concatenator is a notational device for constructing vectors and matrices by concatenation. The rules for the use of a concatenator will be given followed by several examples.

Operations within a concatenator are horizontal concatenation (denoted by a comma) and vertical concatenation (denoted by a number sign). Horizontal concatenation has precedence over vertical concatenation and is performed first whenever both operations appear. Two structures being concatenated must conform, i.e., have the same number of rows for horizontal concatenation and the same number of columns for vertical concatentation. Both of the structures being concatenated must be of the same type, all arrays must be rectangular and the result is also rectangular. As an example of the use of array constructors, consider the following:

A has M rows and N columns (matrix)

B has M rows and 1 column (column vector)

C has 1 row and N column (row vector)

(B, TRANSPOSE(C)) has M rows and 2 columns: $(B\ C^T)$

(A,B) or A,B has M rows and N+1 columns: (A B)

(A)#(C) has M+1 rows and N columns $\begin{pmatrix} A \\ C \end{pmatrix}$

(A,B)#(C,0) has M+1 rows and N+1 columns $\begin{pmatrix} A & B \\ C & 0 \end{pmatrix}$

The above examples of correct usage of the array constructor while the following examples display incorrect usage because of the incompatability of the rows and columns.

(A, C)

(A # B)

An array designator is used to horizontally concatenate several matrices D(J) for J in some index set L. For example L might be a list of basic columns L(1), L(2),...,L(M). Then the basis B is given by

```
B := (A(*,J)  FOR  J  IN  L);
```

Alternatively, it can be written

```
B := (A(*, L(I))  FOR  I  IN (1,...,M));
```

However, it should not be written

```
B := A(*,J)  FOR  J  IN  L;
```

because without the concatenation symbol it is equivalent to

```
FOR  J  IN  L  DO
B := A(*, J);
ENDFOR;
```

which is quite different. Nor should it be written

```
B := (A(*, L(1)),...,A(*, L(M)));
```

because this does not define the running index and $(k,\ldots,\ell)$ in MPL means $(k, k+1,\ldots,\ell)$. Still simplier we can write

```
B := A(*, L);
```

4.0 Statements

All statements in MPL are categorized first by whether or not they are preceeded by a label. All statements are ended by the terminator semi-colon (;).

4.1 Labeled Statements

A label is a means of providing a specific location in a program to which execution control may be transferred. Labels are either a string of digits enclosed in parentheses or can have a name like a variable. A labeled statement consists of a label, followed by a colon followed by an "unlabeled statement" (defined in 4.2) and may be used only once as a label within each storage block. A label can only be referred to later in GO TO statements. Examples:

```
                VAR := X + Y;

UPDATING:  ITERATIONS':= ITERATIONS + 1;
                      :
                GO TO UPDATING;
```

4.2 Unlabeled Statements

Unlabeled statements are of three types--assignment statements, procedure call statements, and keyword statements.

4.2.1 Assignment Statement

Assignment statements are used for transferring data values between data storage locations. The form of a substitution statement is V := AE; where V is any variable as defined in Section 2.1 and AE is any arithmetic expression as defined in Section 3.2. Examples:

```
A := B+C;
S := ARGMIN(V);
A(I,*) := B+C - 3*D;
```

4.2.2 Procedure Call Statement

A procedure call statement transfers execution control to a procedure. When the execution of the procedure is completed, control returns to the statement following the procedure reference. More will be said about procedures in Section 5.1. Examples:

```
PIVOT(M,R,S);
SIMPLEX("IN" A,B,C, "OUT" Z, BV, X_BV);
```

4.2.3 Keyword Statements

Much of the power of MPL lies in the use of keyword statements. Formally, a keyword statement is one which begins with reserve words such as DEFINE, FOR, IF, GO TO, LET, ENDIF, RELEASE, RETURN. The complete list will be found in 3.2.4 in Part III. The keyword indicates to the computer and the programmer what type of action is desired. Some of the keyword statements will be discussed here, the remainder being discussed in Chapter 5 (Statement Blocks).

4.2.3.1 GO TO Statement

A GO TO statement is used to alter the normal sequential flow of control during the execution of a program. The form is GO TO ℓ ; where ℓ is any label as defined in Section 4.1. Example:

```
ITERATE:   I': = I + 1;
                 :
           GO TO ITERATE;
```

4.2.3.2 Simple Conditional Statement (IF)

A simple conditional statement enables one to execute a single statement only

if certain conditions hold, and skip it otherwise. The form is

s IF le;

where le is any logical expression as defined in Section 3.1 and s is an assignment statement. Examples:

S := 0 IF A(*, J) = B;

R := S+T IF Z = 0;

K := R IF U = 0;

L := S IF V $\geq$ 0;

If the logical expressions le is true, the program is executed with s replacing the entire conditional statement. If not true, the program goes to the next statement.

In section 5.4 a compound conditional form is discussed. Its form is

IF le THEN $s_1,\ldots,s_\ell$
OR IF le THEN $s_{\ell+1},\ldots,s_m$
OTHERWISE $s_{m+1},\ldots,s_n$
ENDIF;

4.2.3.3 <u>Simple Iterated Statement</u> (FOR)

A simple iterated statement is used to perform a given statement several times in such a manner that during each execution an iteration index is changed according to a predetermined pattern. The form is

s FOR v IN set;

where v is any variable name as defined in Section 2.1, set is any index set variable as defined in Section 2.1 and s is a statement. s in general depends on v . The first part of the conditioned statement (the FOR phrase) states that the values of an iteration index (v are to range over set). The first cycle through s is executed with the first value of v in set; the second cycle is executed, the second value of v in set, and so forth until the last value of the iteration index has been used in the execution of s. Then control is passed onto the next statement. Example:

```
A(I) := B(I,J) FOR I IN (1,...,M);
```

In Section 5.3 a compound iterated statement is discussed. Its form is

FOR V IN set DO
$s_1, \ldots, s_m$
ENDFOR;

4.2.3.4 Let Statement

The let statement enables one to represent one symbol by another and was introduced into MPL to enhance readability. This statement is similar to a MACRO. It causes modification of the program at compiler time instead of execution time. The let statement will be explained by showing several examples of its use.

```
                    a)  LET M := MATRIX;
                        A := M*B;
is equivalent to        A := MATRIX * B;

                    b)  LET  L(I) := RHS(I)/A(I,S); LET T := (1,...,M);
                        R := ARGMIN(L(T));
is equivalent to        R := ARGMIN(L(J) FOR J IN T);
or equivalent to        R := ARGMIN(RHS(I)/A(I,S)  FOR I IN (1,...,M));
```

```
c) LET  BI := BASIS_INVERSE; LET BC := BASIC_COSTS;
   PI := BC*BI;
```
is equivalent to
```
   PI := BASIC_COSTS*BASIS_INVERSE;
```

Note also in the first example that I is a dummy and that another symbol J was used in its place later on. The form of a let statement is LET v := e where v is a variable and e is an expression.

In the case that let is only used to simplify a single statement, an inverted let or WHERE form can be used.

```
R := ARGMIN(L(J) FOR J IN T)
WHERE  T := (1,...,M);
```

4.2.3.6 Define Statement

Before a variable name may be used in a program the type, structure and storage requirements of the values which it represents must be explicitly or implicitly defined. The only exception to this rule is that an undefined variable may be used as a dummy iteration index or as a dummy variable in a let or where situation. The declaration may be done in two ways. One is to define the variable but not give it any values:

```
DEFINE  V  1  BY  M;
```

The other is to define the variable and assign it values at the same time. In the example below V is a new variable while A and B have been previously defined.

```
V := A + B;
```

Let us now explore the details and meaning of the define statement.

The form of an explicit DEFINE statement is

DEFINE	Variable	Type	Shape	SIZE Dimensions or Domain
	name	ARITHMETIC	RECTANGULAR	m BY n
			DIAGONAL	$(m_1,\ldots,m_2)$ BY $(n_1,\ldots,n_2)$
			UPPER TRIANGULAR	
			LOWER TRIANGULAR	
			SPARSE WITH K NONZEROS	
	name	LOGICAL		
	name	CHARACTER		m
	name	SET		n

Words "ARITHMETIC", "RECTANGULAR" will be understood if type, shape or size descriptors are omitted. Scalar is assured if size description is missing. Let symbols k, m, n, m_1, m_2, n_1, n_2 be any previously defined integers or integer expressions. A matrix "SPARSE WITH K NON-ZEROS" means the matrix has at most k non-zeros. It will be stored as a sparse matrix. A list which has neither row nor column interpretation may be indicated by "(m)" where m is the number of elements. Examples:

1.	DEFINE	E	M BY N;
2.	DEFINE	D, E	DIAGONAL P BY B;
3.	DEFINE	D	(1,...,M) BY (K,...,1);
4.	DEFINE	J;	
5.	DEFINE	M	SPARSE WITH P NONZEROS;
6.	DEFINE	C	1 BY N;
7.	DEFINE	B	M BY 1;
8.	DEFINE	L	CHARACTER;
9.	DEFINE	S	SET;

The form of a domain descriptor is SRL where SRL is a subscript range list, a series of subscript ranges separated by a BY. A subscript range is two arithmetic expressions separated by ,..., . Example of subscript range list: (1,...,M) BY (M+N,...,K). Each subscript range determines the minimum and maximum values of the array's subscripts. The number of subscript ranges in the subscript range list determines the number of dimensions of the storage structure. If the domain is of the form (1,...,M) BY (1,...,N) it is written in Dimension form M BY N or simply M for a one-dimensional list or set. The description shape and size descriptions may appear in any order in a define statement.

The second (and most used) method of defining a variable is implicitly. The form of an implicit define statement is vn := ae; where vn is a variable name as defined in Section 2.1 and ae is an arithmetic expression as defined in Section 3.2. In this version of the define statement the variable name being defined is given the same form, type, and structure as the value of the first arithmetic expression. Examples:

```
M := (A, B)#
     (C, D);

M := (A, B, C);

B := (P(*, BL(I))  FOR  I  IN (1,...,M));

D := E + F*G;  "WHERE  E  AND  F  ARE MATRICES"
```

5.0 Statement Blocks

A program in MPL consists of a sequence of statements (defined in 4.0) and statement blocks. A statement block is a sequence of statements with special initiating and terminating statements. There are four kinds of statement blocks--procedure blocks, storage allocation blocks, conditional blocks and iteration blocks. The entire program is a procedure block. A block can have other blocks imbedded within it, or it may be imbedded in other blocks, but no two blocks partially overlap.

5.1 Procedure Blocks

A procedure is designed to carry out a specific sequence of operations which may be required over and over again. Rather than rewriting the sequence of steps each time, they may be written once in a form which can be utilized whenever needed. It is hoped that a library of procedures written in MPL will be developed, thereby enabling the work of one programmer to be available to others. This will not only speed up the writing of MPL codes, but will also enhance the readability. Later on we will say how to call a procedure in a program.

If one wants to write a procedure (which will later be called by some main routine), the procedure is initiated by a procedure statement, contains a statement sequence, and is terminated by a fini statement. A procedure statement consists of the reserved word PROCEDURE followed by a procedure identifier. The procedure identifier specifies both the procedure name and the local names of the input-output parameters. The form of a procedure identifier is a variable name followed usually by a list of parameters enclosed in a pair of parentheses.

The fini statement is used to mark the end of a procedure write up. In contrast, RETURN is a signal during execution of a program that control is to be passed back to the main routine. This also terminates any storage allocation, iteration, or conditional blocks which were initiated but not explicitly or implicitly terminated within the procedure.

Control is passed to a procedure by either a function or a procedure reference call. A procedure may have several return statements, each one may cause termination during execution. Values are transferred to and from the procedure by means of substitution statements in the input-output section of the procedure identifier. In general, new variables for the main routine may be defined in the output section.

As an example of the use of the return statement in a procedure consider the following routine for checking whether two column vectors are equal.

COMPARE := 0 means A = B.

```
              PROCEDURE  COMPARE(A,B)
      (1):    IF ROW_DIM(A) ¬= ROW_DIM(B)  THEN
                   COMPARE := 1;
                   RETURN;
              OTHERWISE
      (2):         FOR I IN ROW_DOM(A) DO
                   IF A(I) ¬= B(I)  THEN
                          COMPARE := 1;
                          RETURN;
              ENDIF;
              ENDFOR;
                   COMPARE := 0;
      (3):         RETURN;
              ENDIF;
              FINI;
```

Next suppose that in a program we have the following sequence of statements:

```
IF COMPARE(X,Y)=0 THEN GO TO(21); OTHERWISE  GO TO (23); ENDIF;
```

thus if the vector X equals the vector Y in each component, control is transferred to the statement (21), if not, it goes to (23).

5.2. Storage Allocation Blocks, Release Statements

Storage allocation blocks are required for the efficient use of memory core in a computer. To release a symbol and any storage for other use, the statement takes the form:

```
RELEASE  A, B;
```

After much debate, it was decided that in writing mathematical programming codes, block storage allocation was preferable to continual re-allocation.

Release of symbols takes place automatically, however, with subprogram blocks and special release blocks.

All symbols and storage except outputs, generated within a procedure are released wh the procedure returns to the main routine. Hence the same symbols outside the procedure can be used with entirely different meanings.
G in the statement

```
Z := A + G  WHERE  G := INVERSE(M);
```

is treated as a dummy variable locally defined within the block and immediately released. However, in the situation

```
LET G := INVERSE(M);
Z := A + G;
```

the release of G is not possible until the end of a procedure unless by a special

release statement

```
RELEASE G;
```

5.3 Iteration Block

An iteration block is a statement sequence which is repeated a number of times only with an iteration index changed between each execution. As such, this is a generalization of the iterated statement (Section 4.2.3.3). An iteration block is initiated by a for statement, contains a statement sequence, and is terminated by an endfor statement. The for statement (very similar to the for phrase of Section 4.2.3.3) governs the behavior of the iteration by specifying the values for the iteration index. Iteration blocks do not release symbols and storage like a subroutine blocks. **Example:** The form is

FOR v IN set DO

$s_1, \ldots, s_\ell$

ENDFOR;

```
FOR I IN (1,...,M) DO
X(I) := Y(I);
J' := J + 1;
A(*,I) := B(I);
ENDFOR;
```

5.4 Conditional Blocks

Conditional blocks are constructions wherein the program selects between a set of mutually exclusive courses of action. A conditional block is initiated by an if statement and terminated by an endif statement. Or if and otherwise statements allow for the provision of multiple alternatives. This construct is a

generalization of the conditional statement (Section 4.2.3.2). Conditional blocks do not release symbols generated within them. The form is:

```
IF  le  THEN  s_1,...,s_ℓ
OR IF  le  THEN  s_ℓ+1,...,s_m
OTHERWISE  s_m+1,...,s_n
ENDIF;

IF  A = B  THEN GO TO (7);
OR IF  A = C  THEN GO TO (8);
OTHERWISE
        B := A;
ENDIF;
```

The OR IF and OTHERWISE are optional in a conditional block. For example

```
IF  le  THEN  s_1,...,s_ℓ  ENDIF;
```

6.0 Examples of MPL Procedures

```
PROCEDURE SUM(F)
                    "SUMS  A  VECTOR  F  OVER ITS DOMAIN"
                    "ACCUMULATE THE RUNNING SUM IN  S."

        (1):  S := 0;
        (2):  SAME LOCATION (S', S);

                    "S' WILL BE THE UPDATED VALUE OF  S  TO BE STORED IN THE SAME
                    LOCATION AS  S  AND THEREAFTER REFERRED TO AS  S."

        (3):  S' := S + F(I)  FOR  I  IN  DOM(F);

                    "ITERATIVELY ADDS  F(I)  TO  S"

        (4):  SUM := S;
        (5):  RETURN; FINI;

PROCEDURE   MIN_1("IN" F, "OUT" K, M)
                    "K  IS THE FIRST INDEX  I  WHERE  F(I)  TAKES ON ITS MINIMUM
                    VALUE  M  OVER DOMAIN OF  F."
                    "INITIALIZE  K  AND  M"
        (1): K := DOM(F)(1);          "I.E. THE FIRST COMPONENT OF THE SET DOM(F)"
        (2): M := F(K);
        (3): SAME LOCATION (K, K'), (M, M');

                    "K', M', ARE UPDATED VALUES OF  K, M"
```

```
(4): FOR  I  IN DOM(F) DO
         IF  F(I) < M  THEN
               K' := I;
               M' := F(I);
         ENDIF;
     ENDFOR;
(5): RETURN; FINI;
```

```
PROCEDURE  COL_PIVOT (A,P,R);
               "WARNING - MODIFIES  A  AND STORES THE RESULT  A'  IN THE
               SAME LOCATION AS  A."
               "PIVOTS  (A, P)  ON  P(R)  WHERE  A  IS A MATRIX AND  P A
               COLUMN VECTOR, AND RETURNS  A',  THE MODIFIED A PART ONLY."

(1): SAME LOCATION  (A', A);
(2): M := ROW_DIM(A);
(3): LET  T := (1,...,M) AND NOT R;
(4): A'(R, *) := A(R, *)/P(R);
(5): A'(I, *) := A(I, *) - A'(R, *) * P(I)  FOR  I  IN  T;
(6): COL_PIVOT := A';
(7): RETURN;  FINI;
```

```
PROCEDURE  REVISED_SIMPLEX_2("IN" A,D,C,BV, "OUT" STATUS, X,Z,K);
               "REVISED_SIMPLEX_2 IS JUST PHASE 2.
               A = MATRIX,  C = COSTS,  D = RHS, BV = BASIC VARIABLES,
               X = BV VALUES,  Z = OBJECTIVE VALUE,  K  ITERATIONS"
               "THE PROBLEM IS TO FIND MIN Z, X ≥ 0,  AX = D, CX = Z.
               IF  MIN Z  IS FINITE,  STATUS = FINITE, OTHERWISE STATUS =
               INFINITE. IT IS ASSUMED THAT  BV  IS A BASIC FEASIBLE SET
               OF VARIABLES."
```

"INITIALIZATION"

(1): K := C,

(2): STATUS := 'FINITE';

"THE FIRST STEP IS TO SET UP THE INITIAL BASIS WHICH CONSISTS OF THE SET OF BASIC VARIABLE COLUMNS, BV, OF A. THUS BASIS := A(BV). LET G BE THE INVERSE OF THE BASIS. WE ARE INTERESTED IN COMPUTING G AND LATER UPDATING IT."

(3): G := INVERSE(BASIS) WHERE BASIS := A(BV);

"ALSO X, THE VALUES OF THE BASIC VARIABLES, ARE INITIALLY"

(4): X := G * B;

"ITERATIVE LOOP"

"THE COSTS ASSOCIATED WITH BASIC COLUMNS ARE C(BV) - HENCE THE SIMPLEX MULTIPLIERS P ARE GIVEN BY"

(5): P := C(BV) * G;

"LET S DENOTE THE INDEX OF THE COLUMN OF A COMING INTO THE BASIS AND CS = C(S)."

```
(6):  MIN_1("IN" C_P * A, "OUT" S, C_S);

          "WHICH IS THE INDEX (ARGUMENT) OF THE SMALLEST COMPONENT
          OF THE VECTOR OF RELATIVE COSTS  C-P * A."
          "TEST FOR FINITE  MIN Z"

(7): GO TO (16) IF C_S ≥ 0;

          "LET  Y  BE THE REPRESENTATIO
          TERMS OF THE BASIS."

(8): Y := G * A(*, S);

          "LET  R  DENOTE THE INDEX OF THE COLUMN IN THE BASIS TO BE
          REMOVED"

      LET  T := (I IN DOM(Y)|Y(I) > 0);
      IF  T  =  NULL THEN
          STATUS := 'INFINITE';
          GO TO (16);
      ENDIF;

(9):  MIN_1("IN" (X(I)/Y(I) FOR I IN T), "OUT" R, Q);

          "UPDATE  X, G, K, BV  DENOTED BY  X', G', K', BV' "

(10): SAME LOCATION (X, X'), (G, G'), (K, K'), (BV, BV');
(11): K' := K + 1;
```

```
(12): X' := X-Y * Q;
      X'(R) := Q;
(13): G' := COL_PIVOT(G,Y,R);

        "COL_PIVOT PIVOTS (G,Y) ON Y(R)  AND RETURNS MODIFIED  G
        PART."

(14): BV'(R) := S;

        "CHANGE  R-TH  BASIC VARIABLE TO  S."
        "UPDATING COMPLETE, RECYCLE"

(15): GO TO (5);

      "TERMINATION"

(16): Z := C(BV) * X;
(17): RETURN;
(18): FINI;
```

MPL

MATHEMATICAL PROGRAMMING LANGUAGE

PART III

A FORMAL DEFINITION OF MPL

PREPARED BY STEPHEN K. SCHUCK
APRIL 1968

COMMITTEE MEMBERS

RUDOLF BAYER	MICHAEL MCGRATH
JAMES BIGELOW	PAUL PINSKY
GEORGE DANTZIG	STEPHEN SCHUCK
DAVID GRIES	CHRISTOPH WITZGALL

THIS IS THE THIRD OF THREE PARTS:

PART I	A SHORT INTRODUCTION
PART II	A GENERAL DESCRIPTION
PART III	A FORMAL DEFINITION

NOTE: BECAUSE THE DEVELOPMENT OF PARTS I AND II WAS SLIGHTLY OUT OF PHASE WITH THE DEVELOPMENT OF PART III THE READER MAY OBSERVE SOME NOTICEABLE, ALTHOUGH NOT SIGNIFICANT, DESCREPENCIES BETWEEN THEM. THESE DESCREPENCIES ARE DUE TO THE FACT THAT MPL IS NOT YET FULLY DEVELOPED AND MANY IDEAS ARE STILL EXPERIMENTAL.

0-1 ABSTRACT

COMMUNICATION WITH A DIGITAL COMPUTER IS A PROBLEM WHICH HAS OCCUPIED MANY PEOPLE FOR A LONG TIME. IN ORDER TO ALLOW THE COMPUTER TO BE MORE WIDELY USED AS A COMPUTATIONAL TOOL MUCH OF THIS EFFORT HAS GONE INTO DEVELOPING SYSTEMS THROUGH WHICH A PERSON MAY COMMUNICATE HIS DESIRES EVEN THROUGH HE IS NOT FAMILIAR WITH THE SOPHISTICATED AND HIGHLY DETAILED PROGRAMMING LANGUAGES AVAILABLE. THE MATHEMATICAL PROGRAMMING LANGUAGE IS ANOTHER ATTEMPT TO PROVIDE A LANGUAGE IN WHICH THE NON-PROGRAMMER MAY WRITE PROGRAMS. THE VALUE OF THIS WORK LIES IN THE FACT THAT IT IS ORIENTED DIRECTLY TOWARD MATHEMATICAL PROGRAMMING. CONSEQUENTLY CONSIDERABLE EFFORT HAS BEEN MADE TO MAKE MPL LOOK AS MUCH LIKE STANDARD MATHEMATICAL NOTATION AS POSSIBLE.

IT IS HOPED THAT THIS WORK WILL PRODUCE A RIGOROUSLY DEFINED LANGUAGE IN WHICH MATHEMATICAL PROGRAMMERS CAN DESCRIBE ALGORITHMS WHICH WILL AT THE SAME TIME BE EASILY UNDERSTOOD BY OTHER MATHEMATICAL PROGRAMMERS AND MEANINGFUL AND VALID COMPUTER PROGRAMS.

SINCE MPL IS A LANGUAGE INTENDED FOR COMMUNICATION BOTH WITH OTHER INDIVIDUALS AND WITH COMPUTERS, ITS DEVELOPMENT IS AN EFFORT TO PROVIDE A 'READABLE' PROGRAMMING LANGUAGE. HOWEVER, FOR A PROGRAM TO BE READABLE (AN EASY TO USE AND RAPID METHOD FOR TRANSFERRING INFORMATION) IT MUST BE BOTH 'UNDERSTANDABLE' (THE NOTATION IS FAMILIAR OR SELF-EXPLANATORY WITHIN ITS CONTEXT) AND 'COMPREHENDABLE' (THE PARTS OF A PROGRAM MUST INTERRELATE IN A MEANINGFUL MANNER FOR THE PROGRAM READER). IN THIS RESPECT THE EMPHASIS OF MPL IS UPON PROVIDING AN UNDERSTANDABLE LANGUAGE. COMPREHENDABILITY WILL STILL BE THE USER'S RESPONSIBILITY.

0-2 TABLE OF CONTENTS

C-2 TABLE OF CONTENTS (CONTINUED)

0-3 MPL LANGUAGE DESIGN PHILOSOPHY

THE PHILOSOPHY BEHIND THE DESIGN OF THE MATHEMATICAL PROGRAMMING LANGUAGE (HEREAFTER CALLED MPL) IS TO PROVIDE A MAXIMUM OF READABILITY TO THE UNINITIATED. THUS IT CAN HOPEFULLY BE ASSUMED THAT THE USER HAS ONLY A FAMILIARITY WITH THE NOTATON OF CURRENT MATHEMATICAL LITERATURE. AS A RESULT THE LANGUAGE DEFINITION ATTEMPTS TO AVOID ABBREVIATIONS WHICH MAY BE OBSCURE, TO KEEP THE NUMBER OF SPECIAL SYMBOLS TO A MINIMUM, AND TO PROVIDE THE MOST FAMILIAR NOTATION AND FORMATION.

AS MPL DEVELOPED IT BECAME OBVIOUS THAT MANY USEFUL STRUCTURES WERE AVAILABLE IN EXISTING LANGUAGES. AS A RESULT THE READER WHO IS FAMILIAR WITH ALGOL, FORTRAN, PL/I, ETC., WILL ENCOUNTER FAMILIAR FORMS AND PHILOSOPHIES. NO ATTEMPT HAS BEEN MADE TO PARALLEL ANY SINGLE SUCH LANGUAGE, BUT WHERE APPLICABLE TO DEVELOP THE BEST THAT WAS AVAILABLE.

0-4 USE OF THE MANUAL

THE FOLLOWING DISCUSSION IS ORGANIZED SO THAT THE READER MAY FOLLOW THE CONSTRUCTION OF MPL FROM THE MOST ELEMENTARY UP THROUGH THE BROADEST CONCEPTS. THE FINAL SECTION IS A RESUME OF THE FORMAL DEFINITIONS SO THAT THIS PAPER MAY BE USED BOTH FOR INSTRUCTION AND AS A REFERENCE MANUAL. EXAMPLES WILL BE LIBERALLY SPRINKLED AMONG THE DESCRIPTIONS.

THE DEFINITION OF MPL WHICH APPEARS HERE IS AIDED BY THE USE OF A METALINGUISTIC OR LANGUAGE-DESCRIBING LANGUAGE WHICH HAS SEVERAL SPECIAL SYMBOLS.

< >	A PAIR OF BROKEN BRACKETS DELIMITS A PHRASE NAME.
' '	A PAIR OF PRIMES DELIMITS A CHARACTER STRING WHICH APPEARS IN A PHRASE EXACTLY AS IT APPEARS WITHIN THE PRIMES.
::=	READ THIS SYMBOL "IS DEFINED AS". IT SEPARATES THE PHRASE NAME ON THE LEFT FROM THE PHRASE DESCRIPTION ON THE RIGHT.
|	READ THIS SYMBOL "OR". IT SEPARATES MUTUALLY EXCLUSIVE DESCRIPTIONS.

EXAMPLE METALINGUISTIC STATEMENTS

<CHARACTER>::=<LETTER>|<DIGIT>|<SPECIAL CHARACTER>

THIS METALINGUISTIC STATEMENT READS "A CHARACTER IS DEFINED AS A LETTER OR A DIGIT OR A SPECIAL CHARACTER."

<ITERATED STATEMENT>::='IF '<EXPRESSION>','<STATEMENT>

THIS READS "AN ITERATED STATEMENT IS DEFINED AS THE CHARACTERS 'IF ' FOLLOWED BY AN EXPRESSION FOLLOWED BY A COMMA FOLLOWED BY A STATEMENT."

1-1 AN ORGANIZATIONAL OVERVIEW

THE MPL LANGUAGE IS DESIGNED TO FACILITATE THE COMMUNICATION OF MATHEMATICAL PROGRAMMING ALGORITHMS. THE COMPLETE STATEMENT OF AN ALGORITHM IN MPL IS A 'PROGRAM'. A PROGRAM IS COMPOSED OF ONE OR MORE 'PROCEDURES', EACH OF WHICH IS A SEQUENCE OF SEVERAL 'STATEMENTS'. EACH STATEMENT IS MADE UP OF 'RESERVED WORDS' AND 'EXPRESSIONS', THE BASIC BUILDING BLOCKS OF MPL. THESE, FINALLY, ARE COMPOSED OF 'CHARACTERS'.

1-2 THE MPL CHARACTER SET

THE CURRENT VERSION OF MPL IS BASED UPON THE CHARACTER SET OF THE IBM 029 KEYPUNCH. FOR CONVENIENCE THESE CHARACTERS ARE GROUPED INTO THE CATEGORIES OF LETTERS, DIGITS, AND SPECIAL CHARACTERS.

<CHARACTER>::=<LETTER>|<DIGIT>|<SPECIAL CHARACTER>

WHERE THE SPECIFIC CHARACTERS IN EACH CATEGORY ARE GIVEN BY:

<LETTER>::='A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'
|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'

<DIGIT>::='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

<SPECIAL CHARACTER>::='('|')'|'<'|'>'|','|'.'|'+'|'-'|'*'|'/'
|':'|'¬'|'='|'''|'_'|' '|'#'|'@'|'%'|'&'|'?'|'|'|'$'

TWO OTHER CHARACTERS ARE AVAILABLE ON THE 029 KEYPUNCH, BUT ARE NOT INCLUDED IN THE ABOVE CATEGORIES DUE TO THEIR SPECIAL USAGE IN MPL. THESE CHARACTERS ARE

';'	STATEMENT TERMINATOR
'"'	COMMENT DELIMITER

1-3 SOME ELEMENTARY PHRASES

<CHARACTER STRING>::=''|<CHARACTER STRING><CHARACTER>

<DIGIT STRING>::=<DIGIT>|<DIGIT STRING><DIGIT>

<NULL PHRASE>::=''|<NULL PHRASE>' '

THESE PHRASES ARE USED IN SEVERAL PLACES THROUGHOUT THE MANUAL. THE CHARACTER AND DIGIT STRINGS ARE JUST STRINGS OF CHARACTERS OR DIGITS AS THEIR NAMES IMPLY. THE NULL PHRASE INDICATES THAT THE PHRASE WHICH IT DESCRIBES MAY BE OMITTED.

2 EXPRESSIONS

```
<EXPRESSION>::='('<EXPRESSION>')'
          |<NUMBER>
          |'TRUE'|'FALSE'
          |'NULL'
          |'''<CHARACTER STRING>'''
          |<VARIABLE>
          |<PROCEDURE CALL>
          |<COMPUTATIONAL EXPRESSION>
          |<DOMAIN ITEM>
          |<CONCATENATOR>
          |<ARRAY CONSTRICTOR>
          |<SUBSET SPECIFIER>
```

EXPRESSIONS ARE ELEMENTS OF MPL WHICH HAVE 'VALUE'. THEY USUALLY DERIVE THEIR VALUES FROM MANIPULATIONS OF VALUES OF CONSTITUENT PARTS. THE MOST BASIC EXPRESSIONS ARE CONSTANTS WITH FIXED VALUES AND VARIABLES WITH VALUES WHICH MAY CHANGE DURING PROGRAM OPERATION. EACH CONSTANT AND VARIABLE, AND CONSEQUENTLY EACH EXPRESSION, HAS AN ASSOCIATED SET OF ATTRIBUTES WHICH DESCRIBE THE PROPERTIES OF THE VALUE OF THE EXPRESSION.

2-1 EXPRESSION ATTRIBUTES

'TYPE' MPL ALLOWS THE USER TO MANIPULATE VALUES WHICH ARE ARITHMETIC QUANTITIES, LOGICAL OR BOOLEAN QUANTITIES, SETS, OR CHARACTER STRINGS. CONSEQUENTLY THE POSSIBLE VALUES FOR THE TYPE ATTRIBUTE ARE ARITHMETIC, LOGICAL, SET, AND CHARACTER. INITIALLY NO ATTEMPT IS BEING MADE TO IMPOSE THE 'FLOATING POINT' AND 'INTEGER' SUB-CLASSIFICATIONS OF THE ARITHMETIC TYPE ON MPL USERS. INSTEAD IT IS HOPED, PERHAPS IN VAIN, THAT THESE HARDWARE IMPOSED CONVENTIONS MAY BE BYPASSED.

'FORM' IF A VALUE HAS TYPE ARITHMETIC, THEN IT MAY BE EITHER A SCALAR QUANTITY, A VECTOR QUANTITY, OR A MATRIX QUANTITY. CONSEQUENTLY THE POSSIBLE VALUES FOR THE FORM ATTRIBUTE ARE SCALAR, VECTOR, AND MATRIX.

'SHAPE' IF A VALUE HAS TYPE ARITHMETIC, ITS FORM USUALLY HAS A RELATED SHAPE ATTRIBUTE WHICH PROVIDES ADDITIONAL INFORMATION ABOUT THE VALUE'S ORGANIZATION. A SCALAR FORM HAS NO SHAPE ATTRIBUTE. A VECTOR MAY BE EITHER A ROW VECTOR OR A COLUMN VECTOR SO ITS POSSIBLE SHAPES ARE ROW AND COLUMN. MATRICES, NORMALLY RECTANGULAR, ARE GIVEN SHAPES TO CONSERVE STORAGE SPACE BY STORING ONLY SUBSETS OF ELEMENTS. POSSIBLE MATRIX SHAPES ARE RECTANGULAR, UPPER TRIANGULAR, LOWER TRIANGULAR, DIAGONAL, AND SPARSE.

2-2 CONSTANTS

A CONSTANT IS AN EXPRESSION WHICH HAS A FIXED VALUE DETERMINED BY THE NAME OF THE CONSTANT. THERE ARE CONSTANTS OF EACH TYPE.

2-2-1 NUMBERS

```
<NUMBER>::=<NUMBER BASE>|<NUMBER BASE><EXPONENT>

<NUMBER BASE>::=<DIGIT STRING>
          |<DIGIT STRING>'.'
          |'.'<DIGIT STRING>
          |<DIGIT STRING>'.'<DIGIT STRING>

<EXPONENT>::='E'<DIGIT STRING>
          |'E''+'<DIGIT STRING>
          |'E''-'<DIGIT STRING>
```

ESSENTIALLY A NUMBER IS A DIGIT STRING (1-3), POSSIBLY CONTAINING A SINGLE DECIMAL POINT. IF THE NUMBER HAS A VERY LARGE OR A VERY SMALL VALUE SO THAT WRITING IT REQUIRES MANY ZEROS, IT BECOMES WORTHWHILE TO USE THE ABBREVIATED 'SCIENTIFIC NOTATION' PROVIDED BY THE EXPONENT. HERE 'E' MEANS 'TIMES TEN TO THE POWER'. THE SYMBOL '' INDICATES THAT THE SIGN FOLLOWING THE 'E' IS OPTIONAL.

EXAMPLE NUMBERS
2   13.6   2.54   16325   15.6E-03   2E5   .006

2-2-2 LOGICAL CONSTANTS

LOGICAL, BOOLEAN, OR TRUTH VALUED EXPRESSIONS RESULT MOSTLY FROM TESTS ON OTHER QUANTITIES WHICH YIELD THE VALUES TRUE OR FALSE. SINCE THERE ARE ONLY TWO POSSIBLE VALUES FOR ANY LOGICAL EXPRESSION THERE ARE ONLY TWO POSSIBLE LOGICAL CONSTANTS, 'TRUE' AND 'FALSE'.

2-2-3 SET CONSTANTS

SETS IN MPL ARE INTENDED PRIMARILY FOR INDEXING OVER ROWS OR COLUMNS OF MATRICES, ITERATION LOOPS, ETC. AS A RESULT, SET ELEMENTS HAVE WHOLE NUMBER VALUES. THERE ARE NO DUPLICATE ELEMENT VALUES IN SETS. HOWEVER, SINCE SETS MAY CONTAIN A VARIABLE NUMBER OF ELEMENTS, THEY HAVE AN ASSOCIATED SIZE OR NUMBER OF ELEMENTS. THE SINGLE MOST IMPORTANT TEST ON A SET IS THEREFORE WHETHER IT IS EMPTY. THUS THE THE SET CONSTANT 'NULL' IS PROVIDED TO FACILITATE THESE TESTS AND FOR OTHER USES.

2-2-4 CHARACTER CONSTANTS

CHARACTER CONSTANTS HAVE THE FORM '''<CHARACTER STRING>'''.

CHARACTER CONSTANTS WERE ORIGINALLY PROVIDED IN MPL FOR CONVEYING FORMAT INFORMATION TO THE INPUT AND OUTPUT ROUTINES. HOWEVER, WITH ONLY SLIGHT DEVELOPMENT A VERY POWERFUL MANIPULATING CAPABILITY APPEARED. A CHARACTER CONSTANT IS ANY STRING OF CHARACTERS DELINEATED BY A PRIME (SINGLE QUOTE) ON EACH END. A PRIME WITHIN A CHARACTER STRING MUST BE REPRESENTED BY TWO ADJACENT PRIMES, I.E. '' (AS OPPOSED TO A DOUBLE QUOTE ").

EXAMPLE CHARACTER CONSTANTS

```
'1H-,25E13.6'
'HELP,HELP'
'THIS IS THE JONES'' HOUSE'
```

2-3 VARIABLES

<VARIABLE>::=<VARIABLE NAME>|<VARIABLE>'('<SUBSCRIPT LIST>')'

VARIABLES REPRESENT VALUES. JUST AS A VARIABLE NAME IS USED TO REPRESENT AN ENTIRE MATRIX OR VECTOR, VARIABLE NAMES WITH SUBSCRIPTS REPRESENT SPECIFIC ELEMENTS OR SETS OF ELEMENTS OF THESE FORMS. MPL VARIABLES CAN REPRESENT VALUES INDIRECTLY. FOR INSTANCE, IF A REPRESENTS A MATRIX, THE ELEMENTS OF THE MATRIX COULD BE NUMBERS, OR THEY COULD BE POINTERS TO OTHER MATRICES. IN THE LATTER MANNER A(I,J)(K,L) WOULD PICK FROM A(I,J) THE POINTER TO SOME MATRIX FROM WHICH THE (K,L)TH ELEMENT WAS ACTUALLY DESIRED. THE POWER HERE IS THAT THE ELEMENTS OF AN ARITHMETIC MATRIX OR VECTOR NOW MAY BE OTHER ARITHMETIC QUANTITIES, LOGICAL QUANTITIES, SETS, OR CHARACTER STRINGS.

2-3-1 VARIABLE NAMES

```
<VARIABLE NAME>::=<LETTER>
         |<VARIABLE NAME><LETTER>
         |<VARIABLE NAME><DIGIT>
         |<VARIABLE NAME>'_'
         |<VAIRABLE NAME>''''
```

A VARIABLE NAME NAMES A STORAGE STRUCTURE AND THEREBY HAS ALL OF THE ASSOCIATED PROPERTIES OF THE STRUCTURE. IF THE STRUCTURE HAS TYPE ARITHMETIC ITS ELEMENTS MAY BE POINTERS TO OTHER STRUCTURES HAVING OTHER TYPES. A VARIABLE NAME ALWAYS BEGINS WITH A LETTER WHICH MAY BE FOLLOWED BY ANY NUMBER OF LETTERS, DIGITS, UNDERSCORES, OR PRIMES.

EXAMPLE VARIABLE NAMES

```
A    A'    ALPHA36    THIS_IS_A_VARIABLE_NAME    OBJECTIVE_FUNCTION
```

2-3-2 SUBSCRIPTS

SUBSCRIPTS ARE SUBSCRIPT LISTS ENCLOSED IN PARENTHESES.

```
<SUBSCRIPT LIST>::=<SUBSCRIPT ELEMENT>
        |<SUBSCRIPT LIST>','<SUBSCRIPT ELEMENT>

<SUBSCRIPT ELEMENT>::='*'|<EXPRESSION>
```

SUBSCRIPTS ARE USED TO ACCESS SUBSETS OF ELEMENTS OF ARITHMETIC DATA STRUCTURES. THE NUMBER OF SUBSCRIPT ELEMENTS IN A SUBSCRIPT LIST MUST BE EQUAL TO THE NUMBER OF DIMENSIONS OF THE DATA STRUCTURE. THE * USED AS A SUBSCRIPT ELEMENT REFERENCES AN ENTIRE ROW OR COLUMN OF AN ARRAY. THUS A(*,*) = A AND B(*) = B WHERE A AND B ARE A MATRIX AND A VECTOR RESPECTIVELY. VALUES OF EXPRESSIONS USED AS SUBSCRIPT ELEMENTS MUST HAVE EITHER ARITHMETIC OR SET TYPE. IF THE EXPRESSION IS ARITHMETIC IT MUST BE EITHER A SCALAR OR A VECTOR. A SCALAR ACCESSES A SINGLE ELEMENT WHILE A VECTOR ACCESSES A SET OF ELEMENTS. ANY FRACTIONAL PART OF A VECTOR OR SCALAR ELEMENT VALUES IS DROPPED AND ANY VALUES OUTSIDE THE RANGE OF THE SUBSCRIPT ELEMENT ARE IGNORED.

EXAMPLE VARIABLES

```
A(3*A+3,C)   A'(I,J)   B(I)   A'(1,*)   A(ROW_SET,COL_SET)
```

AS MENTIONED IN (2-3) THE ELEMENTS OF AN ARITHMETIC DATA STRUCTURE (VECTOR OR MATRIX) MAY ALSO POINT TO OTHER SUCH QUANTITIES. HENCE 'MATRIX_LIST(K)(I,J)' ACCESSES THE (I,J)TH ELEMENT IN THE MATRIX INDICATED BY THE (K)TH ELEMENT IN 'MATRIX_LIST'. THIS PROCESS MAY BE CONTINUED TO ANY LEVEL, BUT WITH CARE.

2-4 PROCEDURE CALLS

```
<PROCEDURE CALL>::=<VARIABLE NAME>
        |<VARIABLE NAME>'('<EXPRESSION LIST>')'

<EXPRESSION LIST>::=<EXPRESSION>|<EXPRESSION LIST>','<EXPRESSION>
```

A PROCEDURE CALL CALLS A PROCEDURE FROM WITHIN AN EXPRESSION. IT IS ASSUMED THAT THE CALLED PROCEDURE RETURNS A VALUE WHICH CAN BE USED TO EVALUATE THE EXPRESSION IN THE CALLING PROCEDURE.

WHEN A PROCEDURE IS DEFINED (3) ANY VALUES WHICH WILL BE PASSED FROM THE CALLING PROCEDURE AT THE TIME OF THE CALL ARE REPRESENTED BY VARIABLE NAMES IN THE VARIABLE NAME LIST FOLLOWING THE PROCEDURE NAME IN THE DEFINITION. THESE VARIABLES TAKE THE VALUES OF THE EXPRESSIONS IN THE PROCEDURE CALL EXPRESSION LIST IN THE ORDER IN WHICH THEY OCCUR.

THE VALUE OF A PROCEDURE IS DETERMINED IN AN ASSIGNMENT STATEMENT WITHIN THE PROCEDURE IN WHICH THE NAME OF THE PROCEDURE APPEARS ON THE LEFT OF THE ASSIGNMENT SYMBOL (3-2-2).

EXAMPLE PROCEDURE CALLS

```
        PIVOT(A+A',B',I+2,J+R-3)
        SUB(B)
```

2-5 COMPUTATIONAL EXPRESSIONS

```
<COMPUTATIONAL EXPRESSION>::='+'<EXPRESSION>
        |'-'<EXPRESSION>
        |'NOT '<EXPRESSION>
        |<EXPRESSION>'+'<EXPRESSION>
        |<EXPRESSION>'-'<EXPRESSION>
        |<EXPRESSION>'*'<EXPRESSION>
        |<EXPRESSION>'/'<EXPRESSION>
        |<EXPRESSION>'**'<EXPRESSION>
        |<EXPRESSION>'#'<EXPRESSION>
        |<EXPRESSION>' AND '<EXPRESSION>
        |<EXPRESSION>' OR '<EXPRESSION>
        |<EXPRESSION>' IN '<EXPRESSION>
        |<EXPRESSION>' AND NOT '<EXPRESSION>
        |<EXPRESSION>'='<EXPRESSION>
        |<EXPRESSION>'¬='<EXPRESSION>
        |<EXPRESSION>'>'<EXPRESSION>
        |<EXPRESSION>'<'<EXPRESSION>
        |<EXPRESSION>'>='<EXPRESSION>
        |<EXPRESSION>'<='<EXPRESSION>
```

'OPERATORS' MODIFY OR CONNECT 'OPERAND' EXPRESSIONS IN COMPUTATIONAL EXPRESSIONS. ALL COMPUTATIONAL EXPRESSIONS HAVE ONE OF TWO GENERAL FORMS:

```
UNARY    <OPERATOR><R-OPERAND>
BINARY   <L-OPERAND><OPERATOR><R-OPERAND>
```

2-5-1 OPERATOR CLASSES AND ALLOWABLE CONFIGURATIONS

EACH OPERATOR HAS A UNIQUE CONTEXT IN WHICH IT MAY BE USED. THE CONTEXT IS DETERMINED BY THE TYPES OF THE ASSOCIATED OPERANDS. AS A RESULT OPERATORS ARE CLASSED AS 'ARITHMETIC', 'SET', 'ARITHMETIC TEST', 'SET TEST', AND 'LOGICAL'.

THE FOLLOWING TABLE DETERMINES THE TYPES OF OPERANDS ALLOWABLE WITH EACH CLASS OF OPERANDS.

L-OPERAND TYPE	OPERATOR CLASS	R-OPERAND TYPE	RESULT TYPE
ARITHMETIC	ARITHMETIC	ARITHMETIC	ARITHMETIC
SET	SET	SET	SET
ARITHMETIC	ARITHMETIC TEST	ARITHMETIC	LOGICAL
SET	SET TEST	SET	LOGICAL
LOGICAL	LOGICAL	LOGICAL	LOGICAL

2-5-2 OPERATOR DEFINITIONS AND PRECEDENCES

THE OPERATORS WHICH FALL INTO THESE CLASSES AND THEIR MEANINGS ARE SHOWN IN THE FOLLOWING TABLE. SO THAT THE ORDER OF COMPUTATION IN ANY COMPLICATED EXPRESSION WILL BE UNAMBIGUOUS, EACH OPERATOR HAS A PRECEDENCE (INDICATED BY A PRECEDENCE NUMBER) AND OPERATIONS WITH THE HIGHEST PRECEDENCE (NUMBER) ARE PERFORMED FIRST. OPERATORS WITH THE SAME PRECEDENCE NUMBER HAVE EQUAL PRECEDENCE AND ARE PERFORMED FROM LEFT TO RIGHT.

OPERATOR DEFINITION TABLE

OPERATOR	PRECEDENCE	USE	INTERPRETATION
	ARITHMETIC OPERATORS		
'#'	70	BINARY	VERTICAL CONCATENATION
'+'	65	UNARY	NO EFFECT
'-'	65	UNARY	NEGATION
'**'	60	BINARY	EXPONENTIATION
'*'	55	BINARY	MULTIPLICATION
'/'	50	BINARY	DIVISION
'+'	45	BINARY	SUM
'-'	45	BINARY	DIFFERENCE
	SET OPERATORS		
' AND '	40	BINARY	SET INTERSECTION
' OR '	35	BINARY	SET UNION
' AND NOT '	30	BINARY	SET RELATIVE COMPLEMENT
	ARITHMETIC TEST OPERATORS		
'='	25	BINARY	IS EQUAL TO
'¬='	25	BINARY	IS NOT EQUAL TO
'>='	25	BINARY	IS GREATER THAN OR EQUAL TO
'<='	25	BINARY	IS LESS THAN OR EQUAL TO
'>'	25	BINARY	IS STRICTLY GREATER THAN
'<'	25	BINARY	IS STRICTLY LESS THAN
	SET TEST OPERATORS		
' IN '	20	BINARY	IS CONTAINED IN (IS A SUBSET OF)
	LOGICAL OPERATORS		
'NOT '	15	UNARY	LOGICAL NEGATION
' AND '	10	BINARY	LOGICAL INTERSECTION
' OR '	5	BINARY	LOGICAL UNION

2-5-3 SEMANTICS

EACH COMPUTATIONAL EXPRESSION HAS THE FORM
<L-OPERAND><OPERATOR><R-OPERAND>

THIS SECTION DESCRIBES THE RESTRICTIONS PLACED UPON EACH OPERAND AND SOME ADDITIONAL PROPERTIES OF THE RESULTS.

ARITHMETIC OPERATORS

THE CURRENT VERSION OF MPL RESTRICTS ARITHMETIC DATA STRUCTURES TO TWO DIMENSIONS. THIS RESTRICTION ALLOWS CONSIDERABLE IMPLICIT COMPUTING POWER WITHOUT BEING OVERLY RESTRICTIVE FOR MATHEMATICAL PROGRAMMING APPLICATIONS. THUS ALL ARITHMETIC DATA STRUCTURES (EVEN THE CONSTANT 15) CAN BE VISUALIZED AS MATRICES.

OPERATOR	PART	CHARACTERISTICS
'#'	L-OPERAND	ANY ARITHMETIC QUANTITY.
	R-OPERAND	AN ARITHMETIC QUANTITY WITH THE SAME NUMBER OF COLUMNS AS THE L-OPERAND.
	RESULT	THE VERTICAL CONCATENATION OF THE TWO OPERANDS. IT HAS THE SAME NUMBER OF COLUMNS AS EACH OPERAND AND THE NUMBER OF ROWS EQUAL TO THE SUM OF THE NUMBERS OF ROWS IN EACH OPERAND.
'+'	L-OPERAND	NONE.
	R-OPERAND	ANY ARITHMETIC QUANTITY.
	RESULT	SAME AS R-OPERAND.
'-'	L-OPERAND	NONE.
	R-OPERAND	ANY ARITHMETIC QUANTITY.
	RESULT	THE R-OPERAND WITH ALL ELEMENT VALUE SIGNS REVERSED.
'**'	L-OPERAND	ANY ARITHMETIC QUANTITY WITH THE SAME NUMBER OF ROWS AND COLUMNS. THUS THE L-OPERAND MAY BE EITHER A SQUARE MATRIX OR A 'SCALAR'.
	R-OPERAND	MUST BE A SCALAR (ONE ROW AND ONE COLUMN) WITH A NON-NEGATIVE VALUE.
	RESULT	THE L-OPERAND MULTIPLIED BY ITSELF THE NUMBER OF TIMES SPECIFIED BY THE R-OPERAND. IF THE L-OPERAND HAS MORE THAN ONE ROW AND COLUMN ANY FRACTIONAL PORTION OF THE R-OPERAND WILL BE DROPPED. OTHERWISE THE L-OPERAND IS A SCALAR AND ANY POSITIVE VALUES FOR THE R-OPERAND ARE ALLOWED.

2-5-3 SEMANTICS (CONTINUED)

OPERATOR	PART	CHARACTERISTICS
'*'	L-OPERAND	ANY ARITHMETIC QUANTITY.
	R-OPERAND	ANY ARITHMETIC QUANTITY WITH THE SAME NUMBER OF ROWS AS THE L-OPERAND HAS COLUMNS EXCEPT THAT EITHER OPERAND MAY BE A SCALAR.
	RESULT	AN ARITHMETIC QUANTITY WITH THE SAME NUMBER OF ROWS AS THE L-OPERAND AND THE SAME NUMBER OF COLUMNS AS THE R-OPERAND. ELEMENT VALUES ARE THE RESULT OF CONVENTIONAL MATRIX MULTIPLICATION. IF EITHER OPERAND IS A SCALAR THE RESULT HAS THE SAME NUMBER OF ROWS AND COLUMNS AS THE OTHER OPERAND.
'/'	L-OPERAND	ANY ARITHMETIC QUANTITY.
	R-OPERAND	ANY SCALAR ARITHMETIC QUANTITY.
	RESULT	HAS ALL THE PROPERTIES OF THE L-OPERAND EXCEPT THAT ALL ELEMENT VALUES HAVE BEEN DIVIDED BY THE R-OPERAND.
'+'	L-OPERAND	ANY ARITHMETIC QUANTITY.
	R-OPERAND	ANY ARITHMETIC QUANTITY WITH THE SAME NUMBER OF ROWS AND COLUMNS AS THE L-OPERAND.
	RESULT	AN ARITHMETIC QUANTITY WITH THE PROPERTIES OF THE L-OPERAND. ALL POINTERS ARE SET TO ZERO.
'-'	SAME AS '+'(BINARY)	

SET OPERATORS

OPERATOR	PART	CHARACTERISTICS
' AND '	L-OPERAND	ANY SET.
	R-OPERAND	ANY SET.
	RESULT	A SET CONTAINING ONLY THOSE ELEMENTS WHICH APPEARED IN BOTH THE L-OPERAND AND THE R-OPERAND.
' OR '	L-OPERAND	ANY SET.
	R-OPERAND	ANY SET.
	RESULT	A SET CONTAINING ALL ELEMENTS WHICH APPEARED IN EITHER THE L-OPERAND, THE R-OPERAND OR BOTH.
' AND NOT '	L-OPERAND	ANY SET.
	R-OPERAND	ANY SET.
	RESULT	A SET CONTAINING ALL ELEMENTS WHICH APPEARED IN THE L-OPERAND BUT NOT IN THE R-OPERAND.

ARITHMETIC TEST OPERATORS

ARITHMETIC TEST OPERATORS IMPOSE THREE DIFFERENT REQUIREMENTS ON THEIR TWO OPERANDS. TO SATISFY THESE REQUIREMENTS BOTH OPERANDS ARE TREATED AS MATRICES. THESE REQUIREMENTS ARE:

1) THE TWO OPERANDS HAVE THE SAME NUMBER OF ROWS.
2) THE TWO OPERANDS HAVE THE SAME NUMBER OF COLUMNS.
3) THE SPECIFIED RELATIONSHIP HOLDS WITHIN EACH PAIR OF CORRESPONDING (L-OPERAND,R-OPERAND) ELEMENTS.

OPERATOR	PART	CHARACTERISTICS
'='	L-OPERAND	ANY ARITHMETIC QUANTITY.
	R-OPERAND	ANY ARITHMETIC QUANTITY.
	RESULT	A LOGICAL QUANTITY WHICH IS TRUE ONLY IF REQUIREMENTS 1), 2), AND 3) ARE SATISFIED WITH THE EQUALITY RELATIONSHIP.
'¬='	L-OPERAND	ANY ARITHMETIC QUANTITY.
	R-OPERAND	ANY ARITHMETIC QUANTITY.
	RESULT	A LOGICAL QUANTITY WHICH IS FALSE ONLY IF REQUIREMENTS 1), 2), AND 3) ARE SATISFIED USING THE EQUALITY RELATIONSHIP.
'>='	L-OPERAND	ANY ARITHMETIC QUANTITY8
	R-OPERAND	ANY ARITHMETIC QUANTITY.
	RESULT	A LOGICAL QUANTITY WHICH IS TRUE ONLY IF REQUIREMENTS 1), 2), AND 3) ARE SATISFIED USING THE GREATER THAN OR EQUAL RELATIONSHIP. AN ERROR CONDITION EXISTS IF EITHER OF REQUIREMENTS 1) AND 2) IS NOT SATISFIED.
'<='	SAME AS '>=' EXCEPT THAT THE RELATIONSHIP FOR REQUIREMENT 3) IS LESS THAN OR EQUAL.	
'>'	SAME AS '>=' EXCEPT THAT THE RELATIONSHIP FOR REQUIREMENT 3) IS STRICTLY GREATER THAN.	
'<'	SAME AS '>=' EXCEPT THAT THE RELATIONSHIP FOR REQUIREMENT 3) IS STRICTLY LESS THAN.	

2-5-3 SEMANTICS (CONTINUED)

SET TEST OPERATORS

OPERATOR	PART	CHARACTERISTICS
' IN '	L-OPERAND	ANY SET.
	R-OPERAND	ANY SET.
	RESULT	A LOGICAL QUANTITY WHICH IS TRUE ONLY IF ALL ELEMENTS OF THE L-OPERAND ARE ALSO ELEMENTS OF THE R-OPERAND.

LOGICAL OPERATORS

OPERATOR	PART	CHARACTERISTICS
'NOT '	L-OPERAND	NONE.
	R-OPERAND	ANY LOGICAL QUANTITY.
	RESULT	A LOGICAL QUANTITY WHICH IS FALSE IF THE R-OPERAND IS TRUE AND IS TRUE IF THE R-OPERAND IS FALSE.
' AND '	L-OPERAND	ANY LOGICAL QUANTITY.
	R-OPERAND	ANY LOGICAL QUANTITY.
	RESULT	A LOGICAL QUANTITY WHICH IS TRUE ONLY IF BOTH THE L-OPERAND AND THE R-OPERAND VALUES ARE TRUE.
' OR '	L-OPERAND	ANY LOGICAL QUANTITY.
	R-OPERAND	ANY LOGICAL QUANTITY.
	RESULT	A LOGICAL QUANTITY WHICH IS FALSE ONLY IF BOTH THE L-OPERAND AND THE R-OPERAND VALUES ARE FALSE.

2-6 OTHER EXPRESSIONS

MPL CONTAINS CONSTRUCTIONS WHICH ARE NOT PROPERLY CLASSED AS COMPUTATIONAL EXPRESSIONS, BUT WHICH ARE USED TO COMBINE VARIABLES, CONSTANTS, OR MORE COMPLICATED EXPRESSIONS INTO LARGER EXPRESSIONS.

2-6-1 DOMAIN ITEMS

```
<DOMAIN ITEM>::='('<EXPRESSION>',...,'<EXPRESSION>')'
```

DOMAIN ITEMS HAVE VALUES WHICH ARE SETS. THE SETS ARE SPECIFIED BY SPECIFYING THE LOWEST AND HIGHEST VALUED ELEMENTS AND ASSUMING THAT ALL INTERMEDIATE VALUED ELEMENTS ARE IN THE SET. BOTH EXPRESSIONS SHOULD HAVE SCALAR ARITHMETIC VALUES AND ONLY THE WHOLE NUMBER PARTS OF THESE WILL BE USED. THE VALUE OF THE FIRST EXPRESSION SHOULD BE LESS THAN THE SECOND. IF THE EXPRESSION VALUES ARE EQUAL THE SET WILL CONTAIN ONE ELEMENT. IF THE FIRST EXPRESSION IS GREATER THAN THE SECOND THE SET WILL BE EMPTY.

EXAMPLE DOMAIN ITEMS

```
(1,...,M)
(I+J-K,...,L-1)
(HERE,...,THERE)
```

2-6-2 CONCATENATOR

```
<CONCATENATOR>::='('<EXPRESSION LIST>')'
```

A CONCATENATOR HAS AN ARITHMETIC VALUE. IT ALLOWS THE CONSTRUCTION OF ARITHMETIC DATA STRUCTURES BY THE EXPLICIT HORIZONTAL CONCATENATION (ADJACENT PLACEMENT) OF SEVERAL SMALLER STRUCTURES WITH THE SAME NUMBER OF ROWS. THE INDICES OF THE RESULTING STRUCTURE BEGIN AT ONE. VERTICAL CONCATENATION IS ACCOMPLISHED USING THE OPERATOR '#'.

EXAMPLE CONCATENATORS

```
(1,3,4,8,10)
(3*I,5*K,2*J+3,I+J,13,69)
(A,B)
```

2-6-3 ARRAY CONSTRUCTOR

<ARRAY CONSTRUCTOR>::='('<EXPRESSION>' '<FOR PHRASE>')'

AN ARRAY CONSTRUCTOR HAS AN ARITHMETIC VALUE. IT ALLOWS THE CONSTRUCTION OF ARITHMETIC DATA STRUCTURES BY THE IMPLICIT HORIZONTAL CONCATENATION OF SEVERAL EXPRESSION VALUES. THUS ALL EXPRESSIONS BEING CONCATENATED MUST HAVE THE SAME NUMBER OF ROWS. THE FOR-PHRASE (3-2-5-2) GOVERNS THE ITERATIVE PROCESS WHICH PROVIDES VALUES TO BE CONCATENATED.

EXAMPLE ARRAY CONSTRUCTORS

```
        (A(*,I)+B FOR I IN S)
        (B(I) FOR I IN (1,...,N))
        (C(J) FOR J IN S|F(J) >= D)
```

2-6-4 SUBSET SPECIFIER

<SUBSET SPECIFIER>::='('<VARIABLE NAME>' IN '<EXPRESSION>
'|'<EXPRESSION>')'

SUBSET SPECIFIERS PRODUCE SETS. THEY FORM SETS FROM LARGER SETS BY SELECTING ELEMENTS WITH A GIVEN PROPERTY. THE VARIABLE NAME REPRESENTS ELEMENTS SELECTED FROM THE 'PARENT' SET SO THAT THEY MAY BE TESTED FOR THE PROPERTY. THE FIRST EXPRESSION DETERMINES THE PARENT SET AND MUST BE SET VALUED. THE SECOND EXPRESSION TESTS THE PROPERTY AND MUST BE LOGICAL VALUED. ONLY THOSE ELEMENTS IN THE PARENT SET FOR WHICH THE LOGICAL EXPRESSION IS TRUE ARE INCLUDED IN THE NEW SET.

EXAMPLE SUBSET SPECIFIERS

```
        (J IN S|A(J,K)<=R)
        (J IN S | J>=D AND J¬= Y)
```

3 PROGRAM CONSTRUCTION

```
<PROGRAM>::='PROCEDURE '<PROCEDURE IDENTIFIER>
        <STATEMENT SEQUENCE>'FINI'';'
       |<PROGRAM>'PROCEDURE '<PROCEDURE IDENTIFIER>
        <STATEMENT SEQUENCE>'FINI'';'

<PROCEDURE IDENTIFIER>::=<VARIABLE NAME>
        |<VARIABLE NAME>'('<VARIABLE NAME LIST>')'

<VARIABLE NAME LIST>::=<VARIABLE NAME>
        |<VARIABLE NAME LIST>','<VARIABLE NAME>
```

A PROGRAM IN MPL IS A COMPLETE STATEMENT OF AN ALGORITHM AND IS MADE UP OF ONE OR MORE PROCEDURE DEFINITIONS. IT IS ASSUMED THAT THE PROGRAM BEGINS WITH THE FIRST PROCEDURE SO DEFINED. IN THE CURRENT VERSION OF THE LANGUAGE PROCEDURE DEFINITIONS MAY NOT BE NESTED (APPEAR WITHIN OTHER PROCEDURE DEFINITIONS) ALTHOUGH PROCEDURE CALLS MAY BE NESTED TO ANY DEPTH (PROCEDURE A CALLS PROCEDURE B WHICH CALLS PROCEDURE C, ETC.).

PROCEDURE DEFINITIONS BEGIN WITH THE KEYWORD 'PROCEDURE' AND END WITH THE KEYWORD 'FINI'. NOTE THAT PROCEDURE DEFINITIONS HAVE THE SAME GENERAL FORM AS A COMPLEX KEYWORD STATEMENT (3-2-5).

THE PROCEDURE IDENTIFIER PROVIDES NAMES FOR THE PROCEDURE AS WELL AS FOR THE INFORMATION WHICH WILL BE PASSED TO THE PROCEDURE BY A CALLING PROGRAM. WHEN THE PROCEDURE IS CALLED THE PARAMETER EXPRESSIONS (SEE PROCEDURE CALLS (2-4)) ARE EVALUATED AND THESE VALUES ARE USED IN THE CALLED PROCEDURE WHEREVER THEIR REPRESENTATIVE NAMES OCCUR.

EXAMPLE PROGRAM COMPOSED OF TWO PROCEDURES

```
PROCEDURE PROG
   ...
SUB(J,K);
   ...
FINIS;
PROCEDURE SUB(E,F)
   ...
RETURN;
   ...
FINIS;
```

3-1 STATEMENT SEQUENCES

```
<STATEMENT SEQUENCE>::=<STATEMENT>|<STATEMENT SEQUENCE><STATEMENT>
```

A STATEMENT SEQUENCE IS A SEQUENCE OF ONE OR MORE STATEMENTS. THIS CONCEPT IS USEFUL FOR DEFINING PROGRAMS (3) AND COMPLEX KEYWORD STATEMENTS (3-2-5).

3-2 STATEMENTS

```
<STATEMENT>::=<LABEL>':'<STATEMENT>
          |<ASSIGNMENT STATEMENT>
          |<PROCEDURE CALL STATEMENT>
          |<KEYWORD STATEMENT>
```

STATEMENTS IN MPL DETERMINE THE SEQUENCE OF OPERATIONS WHICH MAKES A PROGRAM MEANINGFUL.

3-2-1 LABELS

```
<LABEL>::=<VARIABLE NAME>|'('<DIGIT STRING>')'
```

LABELS ARE EITHER VARIABLE NAMES OR STRINGS OF DIGITS ENCLOSED IN PARENTHESES. SINCE MPL IS WRITTEN IN A FREE FORMAT, A LABEL MUST BE SEPARATED FROM THE FOLLOWING STATEMENT BY A COLON ':'. LABELS MAY ONLY BE REFERENCED BY 'GO TO' STATEMENTS (3-2-4-2).

EXAMPLE LABELED STATEMENTS

```
LABEL:  VAR:=EXP;
LOCATION_B:  VAR2:=EXP2;
(13): VAR3:=EXP3;
```

3-2-2 ASSIGNMENT STATEMENTS

```
<ASSIGNMENT STATEMENT>::=<VARIABLE>':='<EXPRESSION>';'
          |<VARIABLE>':='<EXPRESSION>' '<FOR PHRASE>';'
          |<VARIABLE>':='<EXPRESSION>' ''IF '<EXPRESSION>';'
          |<VARIABLE>':='<EXPRESSION>' WHERE '<SYMBOL SUBSTITUTER>';'
```

ASSIGNMENT STATEMENTS ALTER THE VALUES OF VARIABLES. THE VARIABLE ON THE LEFT OF THE ASSIGNMENT SYMBOL TAKES THE VALUE OF THE EXPRESSION ON THE RIGHT. THIS EXPRESSION MUST HAVE THE SAME TYPE AS THE VARIABLE.

EXAMPLE ASSIGNMENT STATEMENTS

```
          A:=B;
          MATRIX:=(A,B)#
                 (C,C);
          YES_OR_NO:=MATRIX¬=INVERSE(A)
          SET1:=SET2 AND SET3 OR SET4;
```

3-2-2 ASSIGNMENT STATEMENTS (CONTINUED)

THE ASSIGNMENT STATEMENT HAS SEVERAL MODIFIED FORMS WHICH ARE PROVIDED TO MAKE MPL A MORE 'NATURAL' LANGUAGE.

THE ITERATED ASSIGNMENT STATEMENT

THE ITERATED ASSIGNMENT STATEMENT PROVIDES A METHOD FOR ITERATIVELY PERFORMING AN ASSIGNMENT. THIS FORM IS EQUIVALENT TO THE SHORT FORM ITERATED STATEMENT (3-2-5-2). FOR PHRASES ARE ALSO DISCUSSED IN (3-2-5-2).

EXAMPLE ITERATED ASSIGNMENT STATEMENTS

```
        A(P_ROW,J):=A(P_ROW,J)/A(P_ROW,P_COL) FOR J IN COLDOM(A);
         A(I,*):=A(I,*)-A(I,P_COL)*A(P_ROW,*) FOR I IN ROWDOM(A)|
             I¬=P_ROW;
```

CONDITIONED ASSIGNMENT STATEMENT

THE CONDITIONED ASSIGNMENT STATEMENT ALLOWS THE SPECIFICATION OF A CONDITION UNDER WHICH AN ASSIGNMENT WILL OCCUR. THIS FORM IS EQUIVALENT TO THE SHORT FORM OF THE CONDITIONED STATEMENT (3-2-5-1).

EXAMPLE CONDITIONED ASSIGNMENT STATEMENTS

```
        B:=B-A(*,J) IF X(J)=1;
         B(I):=R(I) IF B(I)>=0;
```

THE ASSIGNMENT STATEMENT WITH SYMBOL SUBSTITUTION

THE ASSIGNMENT STATEMENT WITH SYMBOL SUBSTITUTION ALLOWS THE USER TO REDUCE THE APPARENT COMPLEXITY OF EXPRESSIONS BY USING A SINGLE SYMBOL TO REPRESENT A LARGE AND COMPLEX STRING OF CHARACTERS AS DEFINED BY THE SYMBOL SUBSTITUTOR FOLLOWING THE 'WHERE' (SEE (3-2-4-1) FOR A DEFINITION OF SYMBOL SUBSTITUTORS). ONLY A SINGLE SUBSTITUTION IS ALLOWED SINCE THE ';' STATEMENT TERMINATOR ALSO TERMINATES THE STRING TO BE SUBSTITUTED. THIS FORM IS SIMILAR TO USING A 'LET' STATEMENT EXCEPT THAT THE (SYMBOL,CHARACTER STRING) EQUIVALENCE ONLY HOLDS WITHIN THE ASSIGNMENT STATEMENT DEFINING IT.

EXAMPLE ASSIGNMENT STATEMENTS WITH SYMBOL SUBSTITUTION

```
         R:=P+Q WHERE P:=INVERSE((A,B)#(C,D));
```

IMPLICIT DEFINE STATEMENT

IF A VARIABLE FIRST APPEARS AS LEFT MEMBER OF AN ASSIGNMENT STATEMENT WITHOUT ITS TYPE STRUCTURE AND STORAGE REQUIREMENTS HAVING BEEN PREVIOUSLY DECLARED BY A DEFINE STATEMENT (3-2-4-4) THESE REQUIREMENTS ARE DETERMINED BY THE EXPRESSION THAT APPEARS AS RIGHT MEMBER. THE IMPLICIT DEFINE CONCEPT IS UNDER DEVELOPMENT AND WILL NOT BE DISCUSSED FURTHER.

3-2-3 PROCEDURE CALL STATEMENT

```
<PROCEDURE CALL STATEMENT>::=<PROCEDURE CALL>';'
```

A PROCEDURE CALL STATEMENT CALLS A PROCEDURE WHICH DOES NOT RETURN A VALUE (VS. THE PROCEDURE CALL WHICH CALLS A PROCEDURE FROM WITHIN AN EXPRESSION). SINCE THE PROCEDURE CALL STATEMENT APPEARS ALONE (NOT IN AN EXPRESSION), ANY VALUE RETURNED BY THE PROCEDURE IS LOST.

EXAMPLE PROCEDURE CALL STATEMENTS

```
        PIVOT(A,P_ROW,P_COL);
        PROC1(A,B,C,D);
        PROC2(I+J-3*K,J-2,WHAT_NOW,(A,B,C));
```

3-2-4 KEYWORD STATEMENTS

```
<KEYWORD STATEMENT>::=<LET STATEMENT>
        |<GO TO STATEMENT>
        |<RETURN STATEMENT>
        |<DEFINE STATEMENT>
        |<RELEASE STATEMENT>
        |<CONDITIONED STATEMENT>
        |<ITERATED STATEMENT>
        |<BLOCK STATEMENT>
```

EACH KEYWORD STATEMENT BEGINS WITH AN MPL KEYWORD. THESE STATEMENTS ARE DIVIDED INTO SIMPLE AND COMPLEX STATEMENTS. COMPLEX STATEMENTS HAVE SPECIAL BEGINNING AND ENDING SYMBOLS AND CONTAIN OTHER STATEMENTS WITHIN THEM. THIS SECTION DISCUSSES ONLY THE SIMPLE KEYWORD STATEMENTS.

3-2-4-1 LET STATEMENT

```
<LET STATEMENT>::='LET '<SYMBOL SUBSTITUTER>';'
        |'SAME LOCATION ''('<VARIABLE NAME>','<VARIABLE NAME>')'';'

<SYMBOL SUBSTITUTER>::=<VARIABLE NAME>':='<CHARACTER STRING>
        |<VARIABLE NAME>'('<VARIABLE NAME LIST>')'':='<CHARACTER STRING>
```

LET STATEMENTS DIFFER FROM OTHER MPL STATEMENTS BY MODIFYING THE PROGRAM AT TRANSLATION TIME INSTEAD OF EXECUTION TIME. THEY CAN MAKE A PROGRAM EASIER TO WRITE AND/OR MORE READABLE BY ALLOWING THE PROGRAMMER TO REPRESENT CHARACTER STRINGS BY SYMBOLS.

THE TWO PARTS OF A SYMBOL SUBSTITUTER ARE THE CHARACTER STRING (1-3) TO THE RIGHT OF THE ASSIGNMENT SYMBOL AND THE IDENTIFIER TO THE LEFT. THE IDENTIFIER PROVIDES A NAME FOR THE CHARACTER STRING AND, OPTIONALLY, NAMES FOR PARAMETERS. IF THE STRING NAME IS DEFINED WITHOUT PARAMETERS EVERY OCCURRENCE OF THE NAME IN THE FOLLOWING TEXT WILL BE REPLACED BY THE CHARACTER STRING. THE PARAMETERS

3-2-4-1 LET STATEMENT (CONTINUED)

ALLOW MODIFICATION OF THE CHARACTER STRING AT THE TIME OF REPLACEMENT WHEN OCCURRENCES OF THE PARAMETER NAMES IN THE CHARACTER STRING ARE REPLACED WITH THE CHARACTER STRINGS PROVIDED AS PARAMETERS WITH THE STRING NAME. IF COMMAS MUST APPEAR WITHIN THESE PARAMETER CHARACTER STRINGS, TWO MUST BE USED FOR EVERY INTENDED SINGLE OCCURRENCE. THUS (A,B) AS A PARAMETER CHARACTER STRING IN A LET STATEMENT MUST BE WRITTEN (A,,B). WHICH IS TO AVOID HAVING THE COMMA TREATED AS A PARAMETER SEPARATOR. THE SEMICOLON TERMINATES THE CHARACTER STRING AND SO MAY NOT OCCUR WITHIN IT.

AS A RATHER EXTREME EXAMPLE, THE STATEMENT

```
        LET A(C,I) := B(I)*C(J);
```

FOLLOWED BY

```
        D(K):=A(R+F,N);
```

YIELDS

```
        D(K):=B(N)*R+F(J);
```

WHILE THE STATEMENT

```
        LET LOOP(VAR,START,INC,STOP):=FOR VAR:=START STEP INC UNTIL
          STOP DO;
```

FOLLOWED BY

```
        LOOP(I,3*M+K,15,N) A(I):=B(I);ENDFOR;
```

YIELDS

```
        FOR I:=3*J+K STEP 15 UNTIL N DO A(I):=B(I);ENDFOR;
```

CERTAINLY THESE ARE RATHER OBSCURE USES IN A MATHEMATICAL PROGRAMMING LANGUAGE, BUT THEY ARE INCLUDED TO GIVE THE READER AN IDEA OF THE POWER WHICH IS INHERENT IN THIS CONCEPT.

IN A MORE CONVENTIONAL USAGE THE STATEMENT

```
        LET B(T):=A(T,*)*X;
```

FOLLOWED BY

```
        IF B(I)>0, GO TO (5);
```

YIELDS

```
        IF A(I,*)*X>0, GO TO (5);
```

THE FORM USING THE KEYWORD 'SAME LOCATION ' INDICATES AN EQUIVALENCE BETWEEN THE TWO SYMBOLS WITHIN THE PARENTHESES.
A SHORT FORM OF LET STATEMENT USING INVERTED WORD ORDER WITH 'WHERE' INSTEAD OF 'LET', IS DISCUSSED UNDER (3-2-2).

3-2-4-2 GO TO STATEMENT

<GO TO STATEMENT>::='GO TO '<LABEL>';'

GO TO STATEMENTS ALTER THE NORMAL SEQUENTIAL FLOW OF PROGRAM EXECUTION BY TRANSFERRING CONTROL TO THE POINT IN THE PROGRAM INDICATED BY THE LABEL (3-2-1).

EXAMPL- GO TO STATEMENTS

```
        GO TO LOC3;
        GO TO (23);
```

3-2-4-3 RETURN STATEMENT

<RETURN STATEMENT>::='RETURN'';'

THE RETURN STATEMENT RETURNS CONTROL FROM A CALLED PROCEDURE TO ITS CALLING PROCEDURE.

EXAMPLE USE OF THE RETURN STATEMENT IN A PROCEDURE

```
PROCEDURE EQUAL(A,B)
IF DOM(A)¬=DOM(B) THEN
  EQUAL:=FALSE;
  RETURN;
ENDIF;
FOR I IN DOM(A),
  IF A(I)¬=B(I) THEN
    EQUAL:=FALSE;
    RETURN;
  ENDIF;
EQUAL:=TRUE;
RETURN;
FINI;
```

3-2-4-4 DEFINE STATEMENT

<DEFINE STATEMENT>::='DEFINE '<VARIABLE NAME LIST><TYPE PHRASE>
<SHAPE PHRASE><SIZE PHRASE>

<TYPE PHRASE>::=' ARITHMETIC'|' LOGICAL'|' SET'|' CHARACTER'
|<NULL PHRASE>

<SHAPE PHRASE>::=' RECTANGULAR'|' DIAGONAL'|' UPPER TRIANGULAR'
|' LOWER TRIANGULAR'|' ROW'|' COLUMN'|' SPARSE WITH '
<EXPRESSION>' NONZEROS'|<NULL PHRASE>

<SIZE PHRASE>::=<EXPRESSION>' BY '<EXPRESSION>
|<EXPRESSION>|<NULL PHRASE>

BEFORE A VARIABLE NAME MAY BE USED IN A PROGRAM THE TYPE, STRUCTURE, AND STORAGE REQUIREMENTS OF THE VALUES WHICH IT REPRESENTS MUST BE DECLARED. THE ONLY EXCEPTIONS ARE THE VARIABLES USED IN ITERATED STATEMENTS (3-2-5-2) AND ARRAY CONSTRUCTORS (2-6-3), AND SET ELEMENT REPRESENTORS USED IN SUBSET SPECIFIERS (2-6-4). SEE IMPLICIT DEFINE ASSIGNMENT STATEMENT UNDER 3-2-2.
VARIABLE NAME LISTS ARE DEFINED UNDER PROGRAMS (3).

THE TYPE PHRASE DETERMINES WHETHER THE VALUE OF THE VARIABLE IS TO BE TREATED AS AN ARITHMETIC, LOGICAL, SET, OR CHARACTER QUANTITY. IF THIS PHRASE IS OMITTED THE VALUE IS ASSUMED TO BE ARITHMETIC.

THE SHAPE PHRASE MAY ONLY BE USED WHEN DEFINING ARITHMETIC QUANTITIES AND DETERMINES THE STRUCTURE OF SPACE REQUIRED FOR STORING THE DATA AS WELL AS ITS ORGANIZATION. IF THE SHAPE

3-2-4-4 DEFINE STATEMENT (CONTINUED)

PHRASE IS OMITTED THE DEFAULT ASSUMPTIONS ARE:

DIMENSION	DEFAULT SHAPE
2	RECTANGULAR
1	COLUMN
0	NONE

THE MODIFIERS 'RECTANGULAR', 'DIAGONAL', 'UPPER TRIANGULAR', AND 'LOWER TRIANGULAR' ARE ONLY MEANINGFUL WHEN DEFINING TWO DIMENSIONAL QUANTITIES (MATRICES) WHILE THE MODIFIERS 'ROW' AND 'COLUMN' ARE MEANINGFUL ONLY WHEN DEFINING ONE DIMENSIONAL QUANTITIES (VECTORS). THE MODIFIER 'SPARSE' CAN CONSERVE STORAGE WHEN THERE IS A PREDOMINANCE OF ZERO ELEMENTS IN THE ARRAY. THE EXPRESSION IN THE SPARSE MODIFIER MUST BE A SCALAR VALUED ARITHMETIC EXPRESSION IN THAT IT INDICATES THE NUMBER OF ELEMENTS OF THE SPARSE ARRAY WHICH ARE ACTUALLY TO BE KEPT.

THE SIZE PHRASE SPECIFIES THE NUMBER OF DIMENSIONS OF THE VARIABLE AS WELL AS THE RANGES OF THE INDICES ON EACH OF THESE DIMENSIONS. THE EXPRESSIONS IN THE SIZE PHRASE MUST BE EITHER DOMAIN ITEMS (2-6-1) OR SCALAR ARITHMETIC EXPRESSIONS. DOMAIN ITEMS GIVE BOTH THE UPPER AND LOWER BOUND ON THE RANGE OF THE SUBSCRIPT WHILE SCALAR ARITHMETIC EXPRESSIONS DETERMINE ONLY THE UPPER BOUND ON THE SUBSCRIPT RANGE AND A LOWER BOUND OF ONE IS ASSUMED. THE TYPE PHRASE, SHAPE PHRASE, AND SIZE PHRASE MAY APPEAR IN ANY ORDER IN A DEFINE STATEMENT.

EXAMPLE DEFINE STATEMENTS

```
        DEFINE J,K ARITHMETIC;
        DEFINE SET1,SET2,SET3 SET;
        DEFINE STRING1 CHARACTER;
        DEFINE A (1,...,M) BY (1,...,N);
        DEFINE A M BY N;
        DEFINE C N ROW;
        DEFINE SPARSE_A M BY N SPARSE WITH I*N NONZEROS;
```

3-2-4-5 RELEASE STATEMENT

<RELEASE STATEMENT>::='RELEASE '<VARIABLE NAME LIST>';'

THE RELEASE STATEMENT EXPLICITLY RELEASES THE STORAGE ALLOCATED BY OR AFTER THE CORRESPONDING DEFINE STATEMENT(3-2-4-4). IT IS IMPROPER TO RELEASE A VARIABLE WHICH WAS DEFINED OUTSIDE OF THE CURRENT BLOCK (3-2-5-3). RELEASE STATEMENTS REFERENCEING VARIABLE NAMES WHICH HAVE NOT BEEN DEFINED OR HAVE ALREADY BEEN RELEASED ARE IGNORED. THE RELEASE STATEMENT ALSO IMPLICITLY RELEASES ALL STORAGE WHICH WAS DEFINED AFTER ANY VARIABLE IN THE NAME LIST (SEE (3-2-5-3) FOR AN EXAMPLE).

EXAMPLE RELEASE STATEMENTS

```
        RELEASE A;
        RELEASE A,B,C,D,R,T;
```

3-2-5 COMPLEX KEYWORD STATEMENTS

THE FOLLOWING SECTION DISCUSSES COMPLEX KEYWORD STATEMENTS. THESE STATEMENTS ALL HAVE THE FORM

```
        <INTRODUCTION><STATEMENT SEQUENCE><TERMINATION>
```

3-2-5-1 CONDITIONED STATEMENT

```
<CONDITIONED STATEMENT>::='IF '<EXPRESSION>','<STATEMENT>
        |'IF '<EXPRESSION>' THEN '<STATEMENT SEQUENCE>
        <OR IF SEQUENCE><OTHERWISE PHRASE>'ENDIF'';'

<OR IF SEQUENCE>::=<NULL PHRASE>
        |<OR IF SEQUENCE>'OR IF '<EXPRESSION>' THEN '
        <STATEMENT SEQUENCE>

<OTHERWISE PHRASE>::='OTHERWISE '<STATEMENT SEQUENCE>|<NULL PHRASE>
```

A CONDITIONED STATEMENT ALLOWS THE USER TO SELECT CONDITIONS UNDER WHICH STATEMENT(S) WILL BE EXECUTED. THE SHORT FORM IS USED ONLY WHEN A CONDITION GOVERNS THE EXECUTION OF A SINGLE STATEMENT. THE LONG FORM ALLOWS THE TESTING OF SEVERAL MUTUALLY EXCLUSIVE CONDITIONS. WHEN A CONDITION IS SATISFIED THE STATEMENTS FOLLOWING THE TEST ARE EXECUTED AND CONTROL PASSES TO THE END OF THE STATEMENT. THE EXPRESSIONS FOLLOWING THE KEYWORD 'IF' AND THE KEYWORD 'OR IF' ARE LOGICAL VALUED. SPECIFICALLY THE LOGICAL EXPRESSION FOLLOWING THE 'IF' IS EVALUATED AND IF TRUE THE FOLLOWING STATEMENT SEQUENCE IS EXECUTED AND CONTROL THEN PASSES TO THE 'ENDIF'. IF THE EXPRESSION IS FALSE THE EXPRESSION IN THE NEXT FOLLOWING 'OR IF' IS EVALUATED WITH THE SAME ACTIONS. IF AN 'OTHERWISE' IS ENCOUNTERED ALL STATEMENTS IMMEDIATELY FOLLOWING THE 'OTHERWISE' ARE EXECUTED.

EXAMPLE CONDITIONED STATEMENTS

```
        IF Z¬=0 , GO TO NON_ZERO;
        IF A(*,J)=B, A(*,J):=A(*,K);
        IF A=B THEN
          GO TO A_EQUAL_B;
        OR IF A=C THEN
          GO TO A_NE_B_BUT_EQ_C;
        OR IF J¬=K AND N>3*P THEN
          R:=N;
        OTHERWISE
          B:=A;
          C:=A;
          GO TO NO_GOOD;
        ENDIF;
```

SEE ALSO CONDITIONED ASSIGNED STATEMENT UNDER (3-2-2) WHERE A SHORT-IF FORM IN INVERTED ORDER IS DISCUSSED.

3-2-5-2 ITERATED STATEMENT

```
<ITERATED STATEMENT>::=<FOR PHRASE>','<STATEMENT>
        |<FOR PHRASE>' DO '<STATEMENT SEQUENCE>'ENDFOR'';'

<FOR PHRASE>::='FOR '<VARIABLE NAME>' IN '<EXPRESSION>
        |'FOR '<VARIABLE NAME>' IN '<EXPRESSION>'|'<EXPRESSION>
        |'FOR '<VARIABLE NAME>':='<EXPRESSION>' STEP '
        <EXPRESSION>' UNTIL '<EXPRESSION>
```

THE FOR PHRASE GOVERNS THE INDEXING OF AN ITERATION. ONE OF THE TWO FORMS INDICATES AN INDEXING OVER ELEMENTS OF A SET, NAMES THE INDEX, SPECIFIES THE SET, AND ALLOWS ELEMENTS OF THE SET TO BE SELECTIVELY DISCARDED. ON EACH CYCLE OF THE ITERATION THE INDEX TAKES ON A NEW VALUE FROM THE SET. THIS INDEX MAY BE USED TO AFFECT STATEMENTS WITHIN THE SCOPE OF THE ITERATION. SELECTIVE DISCARDING OF ELEMENTS IS PERFORMED BY THE OPTIONAL EXPRESSION FOLLOWING THE 'SUCH THAT' SYMBOL ('|'). HENCE THE INDEX VARIABLE AND FIRST EXPRESSION MUST BE SCALAR ARITHMETIC QUANTITIES, THE SECOND EXPRESSION MUST BE SET VALUED, AND THE OPTIONAL THIRD EXPRESSION MUST BE LOGICAL VALUED.

THE SECOND FORM SPECIFIES THE INDEXING IN A MORE CONVENTIONAL MANNER IN WHICH THE INDEX IS GIVEN A STARTING VALUE FOR THE FIRST CYCLE AND THAT VALUE IS INCREMENTED BY THE STEP ON EACH SUCCESSIVE CYCLE. THE TERMINAL CONDITION IS TESTED ON EVERY CYCLE BEFORE ANY ENCLOSED STATEMENTS ARE EXECUTED. EXECUTION OF THESE STATEMENTS OCCURS AS LONG AS THE CONDITION IS NOT SATISFIED. THUS THE VARIABLE NAME AND THE FIRST TWO EXPRESSIONS MUST BE SCALAR ARITHMETIC QUANTITIES WHILE THE TERMINAL CONDITION EXPRESSION MUST BE LOGICAL VALUED. THIS SECOND FORM DOES NOT PROVIDE AN ADDITIONAL TEST FOR SCREENING INDICES.

EXAMPLE ITERATED STATEMENTS

```
        FOR I IN (1,...,M), A(I):=B(I,J);
        FOR I IN SET1 | I¬=P, FOR J IN SET2, A(I,J):=C.;
        FOR I IN SET2 OR SET3 | B(I)>=0 DO
          B(I):=-B(I);
          R:=R+1;
        ENDFOR;
        FOR K:=1 STEP 2 UNTIL K>=N, A(K):=B(K);
```

SEE ALSO ITERATED ASSIGNMENT STATEMENT UNDER (3-2-2) WHERE THE ABOVE FIRST (SHORT) FORM IS DISCUSSED IN INVERTED ORDER.

3-2-5-3 BLOCK STATEMENT

<BLOCK STATEMENT>::='BLOCK '<STATEMENT SEQUENCE>'ENDBLOCK'';'

ALLOCATION AND HANDLING OF STORAGE IS A MAJOR PROBLEM IN MPL SINCE IT WILL BE USED TO SOLVE PROBLEMS INVOLVING LARGE AMOUNTS OF DATA. THE BLOCK STATEMENT ALLOWS THE PROGRAMMER TO DIVIDE HIS PROCEDURES INTO BLOCKS WITHIN WHICH HE CAN ALLOCATE (DEFINE (3-2-4-4)) STORAGE. THIS SPACE IS AUTOMATICALLY RELEASED WHEN CONTROL LEAVES THE BLOCK. IN ADDITICN STORAGE MAY BE EXPLICITLY RELEASED (3-2-4-5) ELSEWHERE IN THE BLOCK IN WHICH IT WAS DEFINED, BUT IN NO OTHER BLOCK. IN THIS CASE STORAGE IS RELEASED IN AN ORDER OPPOSITE THAT OF DEFINITION. THUS THE SEQUENCE

```
DEFINE A;
DEFINE B;
 . . .
RELEASE A;
```

CAUSES BOTH B AND A TO BE RELEASED IN THAT ORDER. NOTICE THAT A PROCEDURE IS AN IMPLIED BLOCK STATEMENT.

EXAMPLE BLOCK STATEMENTS

```
BLOCK
  DEFINE MATRIX M+1 BY N+1;
  MATRIX:=(A,B)*
          (C,Z);
ENDBLOCK;  "EVEN THOUGH IT IS ASSUMED THAT A, B, C,
            AND Z ARE DEFINED OUTSIDE THE BLOCK, THIS
            STATEMENT PRODUCES NO USABLE RESULTS"
```

4 INPUT / OUTPUT

VERY LITTLE WORK HAS YET BEEN DONE ON THIS SECTION. IT IS CURRENTLY THROUGHT THAT MANY IDEAS WILL BE ADOPTED FROM LANGUAGES SUCH AS ALGOL, FORTRAN, OR PL/I.

5 LIBRARY PROCEDURES

THIS SECTION DESCRIBES THE USE OF SEVERAL PROCEDURES WHICH ARE PROVIDED IN THE MPL LIBRARY. REFERENCES TO THESE PROCEDURES ALL HAVE THE FORM F(P) WHERE F REPRESENTS THE NAME OF THE PROCEDURE AND P REPRESENTS A LIST OF PARAMETERS. WHERE INDICATED THESE PROCEDURES RETURN VALUES WITH TYPE, SHAPE, AND FORM AS DESCRIBED BELOW.

ARGMAX(VECTOR)
VECTOR AN ARITHMETIC EXPRESSION WITH A VECTOR VALUE.
VALUE THE SCALAR ARITHMETIC INDEX OF THE FIRST OCCURRING MAXIMUM VALUED ELEMENT OF 'VECTOR'.

ARGMIN(VECTOR)
VECTOR ANY VECTOR VALUED ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC INDEX OF THE FIRST OCCURRING MINIMUM VALUED ELEMENT OF 'VECTOR'.

COLDIM(MATRIX)
MATRIX ANY ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN THE RANGE OF THE SECOND SUBSCRIPT OF 'MATRIX'. THIS FUNCTION IS INTENDED FOR FINDING THE NUMBER OF COLUMNS IN A MATRIX, SO IF 'MATRIX' IS A VECTOR OR SCALAR EXPRESSION, V := 1.

DIM(VECTOR)
VECTOR ANY ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN THE RANGE OF THE FIRST OR ONLY SUBSCRIPT OF 'VECTOR'. IF 'VECTOR' IS MATRIX VALUED THIS PROCEDURE IS EQUIVALENT TO ROWDIM. IF 'VECTOR' IS SCALAR VALUED, V := 1.

IDENTITY(RANK)
RANK THE SCALAR ARITHMETIC RANK OF THE SQUARE IDENTITY MATRIX WHICH IS THE VALUE OF THE PROCEDURE.
VALUE AN IDENTITY MATRIX WITH 'RANK' ROWS AND COLUMNS.

INVERSE(MATRIX)
MATRIX A SQUARE, NON-SINGULAR, MATRIX VALUED ARITHMETIC EXPRESSION.
VALUE THE INVERSE OF 'MATRIX'.

MAX(VECTOR)
VECTOR A VECTOR VALUED ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC VALUE OF THE MAXIMUM VALUED ELEMENT OF 'VECTOR'.

MIN(VECTOR)
VECTOR ANY VECTOR VALUED ARITHMETIC EXPRESSION.
VALUE THE SCALAR ARITHMETIC VALUE OF THE MINIMUM VALUED ELEMENT OF 'MATRIX'. ALL POINTERS ARE IGNORED.

5 LIBRARY PROCEDURES (CONTINUED)

```
ONES(ROWS,COLUMNS)
ROWS      THE SCALAR ARITHMETIC NUMBER OF ROWS IN V.
COLUMNS   THE SCALAR ARITHMETIC NUMBER OF COLUMNS IN V.
VALUE     A MARTIX OF ONES WITH 'ROWS' ROWS AND 'COLUMNS' COLUMNS.

ROWDIM(MATRIX)
MATRIX    ANY ARITHMETIC EXPRESSION.
VALUE     THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN THE RANGE
          OF THE FIRST SUBSCRIPT OF 'MATRIX'.  THIS PROCEDURE IS
          INTENDED FOR FINDING THE NUMBER OF ROWS IN A MATRIX,
          BUT IS EQUIVALENT TO DIM(VECTOR) IF 'MATRIX' IS ACTUALLY
          VECTOR VALUED.  IF 'MATRIX' IS SCALAR VALUED, V:= 1.

SUM(VECTOR)
VECTOR    A VECTOR VALUED ARITHMETIC EXPRESSION.
VALUE     THE SCALAR ARITHMETIC SUM OF THE ELEMENTS OF 'VECTOR'.

TRANSPOSE(MATRIX)
MATRIX    ANY ARITHMETIC EXPRESSION.
VALUE     THE TRANSPOSE OF 'MATRIX'.  IF 'MATRIX' HAS 'M' ROWS AND
          'N' COLUMNS THEN V HAS 'N' ROWS AND 'M' COLUMNS.

UNIT(SIZE,INDEX)
SIZE      THE SCALAR ARITHMETIC NUMBER OF ELEMENTS IN VECTOR 'V'.
INDEX     THE SCALAR ARITHMETIC SUBSCRIPT OF THE SINGLE ONE VALUED
          ELEMENT IN 'V'.  HERE 1 <= INDEX <= SIZE.
VALUE     AN ARITHMETIC COLUMN VECTOR WITH SUBSCRIPT RANGE
          (1,...,SIZE) WHICH HAS ALL ZERO ELEMENTS EXCEPT FOR THE
          SINGLE ONE ELEMENT IN THE INDEX'TH POSITION.

ZEROS(ROWS,COLUMNS)
ROWS      THE SCALAR ARITHMETIC NUMBER OF ROWS IN 'V'.
COLUMNS   THE INTEGER SCALAR NUMBER OF COLUMNS IN 'V'.
VALUE     A MATRIX OF ZEROS WITH 'ROWS' ROWS AND 'COLUMNS' COLUMNS.

ALSO
          SIZE...SCALAR ARITHMETIC VALUED PROCEDURE FOR FINDING THE
               NUMBER OF ELEMENTS IN A SET.
          SET...SET VALUED PROCEDURE FOR CONVERTING ARITHMETIC
               QUANTITIES TO SETS.
          DOM...SET VALUED PROCEDURE FOR INDEXING OVER VECTOR ELEMENTS.
          ROWDOM...SET VALUED PROCEDURE FOR INDEXING OVERMATRIX ROWS.
          COLDOM...SET VALUED PROCEDURE FOR INDEXING OVER MATRIX COLUMNS.
```

6 PROGRAM FORMATION MECHANICS

6-1 CARD FORMAT

MPL USES A 'FREE FORMAT' STYLE WHICH MEANS THAT STATEMENTS MAY BE STRUNG ONE IMMEDIATELY AFTER THE OTHER, ONLY SEPARATED BY THE ';' TERMINATORS. THUS MUCH OF THE RESPONSIBILITY FOR AN AESTHETIC AND READABLE PROGRAM RESTS ON THE WRITER.

WHEN COMMUNICATING THE PROGRAM TO THE COMPUTER ON PUNCH CARDS THE PROGRAM 'TEXT' MUST BE CONFINED TO COLUMNS 1 THROUGH 72. COLUMNS 73 THROUGH 80 MAY BE USED FOR IDENTIFICATION SINCE THEY WILL BE IGNORED. THIS IS A COMMON PROGRAMMING CONVENTION.

6-2 USE OF BLANKS

BLANKS ARE USED AS DELIMITERS IN MPL AND ARE REQUIRED WHERE SPECIFIED IN THE VARIOUS DEFINITIONS. IN ADDITION THEY MAY BE INSERTED BETWEEN ANY TWO SYMBOLS (ITEMS ENCLOSED IN PRIMES IN THE METALANGUAGE DEFINITION) BUT MAY NOT APPEAR WITHIN VARIABLE NAMES OR KEY WORDS EXCEPT WHERE SPECIFIED.

WHEREVER A BLANK IS ALLOWED OR REQUIRED ANY NUMBER OF MULTIPLE BLANKS IS ALLOWED.

6-3 COMMENTS

COMMENTS MAY BE PLACED ANYWHERE IN AN MPL PROGRAM SINCE THEY ARE COMPLETELY IGNORED BY THE COMPUTER. THEY ARE DELIMITED ON BOTH ENDS BY A QUOTE (")(THIS IS NOT A DOUBLE PRIME ('')). OBVIOUS CARE MUST BE TAKEN TO INSURE THAT THE TERMINAL QUOTE APPEARS IN ITS PROPER PLACE.

7 RESUME OF DEFINITIONS

```
<ARRAY CONSTRUCTOR>::='('<EXPRESSION>' '<FOR PHRASE>')'
                                                        2-6-3
<ASSIGNMENT STATEMENT>::=<VARIABLE>':='<EXPRESSION>';'
         |<VARIABLE>':='<EXPRESSION>' '<FOR PHRASE>';'
         |<VARIABLE>':='<EXPRESSION>' ''IF '<EXPRESSION>';'
         |<VARIABLE>':='<EXPRESSION>' WHERE '<SYMBOL SUBSTITUTER>';'
                                                        3-2-2
<BLOCK STATEMENT>::='BLOCK '<STATEMENT SEQUENCE>'ENDBLOCK'';'
                                                        3-2-5-3
<CHARACTER>::=<LETTER>|<DIGIT>|<SPECIAL CHARACTER>
                                                        1-2
<CHARACTER STRING>::=''|<CHARACTER STRING><CHARACTER>
                                                        1-3
<COMPUTATIONAL EXPRESSION>::='+'<EXPRESSION>
         |'-'<EXPRESSION>
         |'NOT '<EXPRESSION>
         |<EXPRESSION>'+'<EXPRESSION>
         |<EXPRESSION>'-'<EXPRESSION>
         |<EXPRESSION>'*'<EXPRESSION>
         |<EXPRESSION>'/'<EXPRESSION>
         |<EXPRESSION>'**'<EXPRESSION>
         |<EXPRESSION>'#'<EXPRESSION>
         |<EXPRESSION>' AND '<EXPRESSION>
         |<EXPRESSION>' OR '<EXPRESSION>
         |<EXPRESSION>' IN '<EXPRESSION>
         |<EXPRESSION>' AND NOT '<EXPRESSION>
         |<EXPRESSION>'='<EXPRESSION>
         |<EXPRESSION>'¬='<EXPRESSION>
         |<EXPRESSION>'>'<EXPRESSION>
         |<EXPRESSION>'<'<EXPRESSION>
         |<EXPRESSION>'>='<EXPRESSION>
         |<EXPRESSION>'<='<EXPRESSION>
                                                        2-5
<CONCATENATOR>::='('<EXPRESSION LIST>')'
                                                        2-6-2
<CONDITIONED STATEMENT>::='IF '<EXPRESSION>','<STATEMENT>
         |IF '<EXPRESSION>' THEN '<STATEMENT SEQUENCE>
         <OR IF SEQUENCE><OTHERWISE PHRASE>'ENDIF'';'
                                                        3-2-5-1
<DEFINE STATEMENT>::='DEFINE '<VARIABLE NAME LIST><TYPE PHRASE>
         <SHAPE PHRASE><SIZE PHRASE>';'
                                                        3-2-4-4
<DIGIT>::='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
                                                        1-2
<DIGIT STRING>::=<DIGIT>|<DIGIT STRING><DIGIT>
                                                        1-3
<DOMAIN ITEM>::='('<EXPRESSION>',....,'<EXPRESSION>')'
                                                        2-6-1
<EXPONENT>::=<DIGIT STRING>
         |'E''+'<DIGIT STRING>
         |'E''-'<DIGIT STRING>
                                                        2-2-1
```

7 RESUME OF DEFINITIONS (CONTINUED)

```
<EXPRESSION>::='('<EXPRESSION>')'
          |<NUMBER>
          |'TRUE'|'FLASE'
          |'NULL'
          |'''<CHARACTER STRING>'''
          |<VARIABLE>
          |<PROCEDURE CALL>
          |<COMPUTATIONAL EXPRESSION>
          |<DOMAIN ITEM>
          |<CONCATENATOR>
          |<ARRAY CONSTRUCTOR>
          |<SUBSET SPECIFIER>
                                                        2
<EXPRESSION LIST>::=<EXPRESSION>|<EXPRESSION LIST>','<EXPRESSION>
                                                        2-4
<FOR PHRASE>::='FOR '<VARIABLE NAME>' IN '<EXPRESSION>
          |'FOR '<VARIABLE NAME>' IN '<EXPRESSION>'|'<EXPRESSION>
          |'FOR '<VARIABLE NAME>':='<EXPRESSION>' STEP '
          <EXPRESSION>' UNTIL '<EXPRESSION>
                                                        3-2-5-2
<GO TO STATEMENT>::='GO TO '<LABEL>';'
                                                        3-2-4-2
<ITERATED STATEMENT>::=<FOR PHRASE>','<STATEMENT>
          |<FOR PHRASE>' DO '<STATEMENT SEQUENCE>'ENDFOR'';'
                                                        3-2-5-2
<KEYWORD STATEMENT>::=<LET STATEMENT>
          |<GO TO STAGEMENT>
          |<RETURN STATEMENT>
          |<DEFINE STATEMENT>
          |<RELEASE STATEMENT>
          |<CONDITIONED STATEMENT>
          |<ITERATED STATEMENT>
          |<BLOCK STATEMENT>
                                                        3-2-4
<LABEL>::=<VARIABLE NAME>|'('<DIGIT STRING>')'
                                                        3-2-1
<LET STATEMENT>::='LET '<SYMBOL SUBSTITUTER>';'
          |'SAME LOCATION ''('<VARIABLE NAME>','<VARIABLE NAME>')'';'
                                                        3-2-4-1
<LETTER>::='A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'
          |'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'
                                                        1-2
<NULL PHRASE>::=''|<NULL PHRASE>' '
                                                        1-3
<NUMBER>::=<NUMBER BASE>|<NUMBER BASE><EXPONENT>
                                                        2-2-1
<NUMBER BASE>::=<DIGIT STRING>
          |<DIGIT STRING>'.'
          |'.'<DIGIT STRING>
          |<DIGIT STRING>'.'<DIGIT STRING>
                                                        2-2-1
```

7 RESUME OF DEFINITIONS (CONTINUED)

```
<OR IF SEQUENCE>::=<NULL PHRASE>
        |<OR IF SEQUENCE>'OR IF '<EXPRESSION>' THEN '
        <STATEMENT SEQUENCE>
                                                        3-2-5-1
<OTHERWISE PHRASE>::='OTHERWISE '<STATEMENT SEQUENCE>
        |<NULL PHRASE>
                                                        3-2-5-1
<PROCEDURE CALL>::=<VARIABLE NAME>
        |<VARIABLE NAME>'('<EXPRESSION LIST>')'
                                                        2-4
<PROCEDURE CALL STATEMENT>::=<PROCEDURE CALL>';'
                                                        3-2-3
<PROCEDURE IDENTIFIER>::=<VARIABLE NAME>
        |<VARIABLE NAME>'('<VARIABLE NAME LIST>')'
                                                        2-4
<PROGRAM>::='PROCEDURE '<PROCEDURE IDENTIFIER>
        STATEMENT SEQUENCE>'FINI'';'
        |<PROGRAM>'PROCEDURE '<PROCEDURE IDENTIFIER>
        <STATEMENT SEQUENCE>'FINI'';'
                                                        3
<RELEASE STATEMENT>::='RELEASE '<VARIABLE NAME LIST>';'
                                                        3-2-4-5
<RETURN STATEMENT>::='RETURN'';'
                                                        3-2-4-3
<SHAPE PHRASE>::=' RECTANGULAR'|' DIAGONAL'|' UPPER TRIANGULAR'
        |' LOWER TRIANGULAR'|' ROW'|' COLUMN'|' SPARSE WITH '
        <EXPRESSION>' NONZEROS'|<NULL PHRASE>
                                                        3-2-4-4
<SIZE PHRASE>::=<EXPRESSION>' BY '<EXPRESSION>
        |<EXPRESSION>|<NULL PHRASE>
                                                        3-2-4-4
<SPECIAL CHARACTER>::='('|')'|'<'|'>'|','|'.'|'+'|'-'|'*'|'/'
        |':'|'¬'|'''|'_'|'|'|'#'|'@'|'%'|'&'|'?'|'$'
                                                        1-2
<STATEMENT>::=<LABEL>':'<STATEMENT>
        |<ASSIGNMENT STATEMENT>
        |<PROCEDURE CALL STATEMENT>
        |<KEYWORD STATEMENT>
                                                        3-2
<STATEMENT SEQUENCE>::=<STATEMENT>|<STATEMENT SEQUENCE><STATEMENT>
                                                        3-1
<SUBSCRIPT ELEMENT>::='*'|<EXPRESSION>
                                                        2-3-2
<SUBSCRIPT LIST>::=<SUBSCRIPT ELEMENT>
        |<SUBSCRIPT LIST>','<SUBSCRIPT ELEMENT>
                                                        2-3-2
<SUBSET SPECIFIER>::='('<VARIABLE NAME>' IN '<EXPRESSION>
        '|'<EXPRESSION>')'
                                                        2-6-4
<SYMBOL SUBSTITUTER>::=<VARIABLE NAME>':='<CHARACTER STRING>
        |<VARIABLE NAME>'('<VARIABLE NAME LIST>')'':='<CHARACTER STRING>
                                                        3-2-4-1
<TYPE PHRASE>::=' ARITHMETIC'|' LOGICAL'|' SET'|' CHARACTER'
        |<NULL PHRASE>
                                                        3-2-4-4
```

7 RESUME OF DEFINITIONS (CONTINUED)

```
<VARIABLE>::=<VARIABLE NAME>|<VARIABLE>'('<SUBSCRIPT LIST>')'
                                                        2-3
<VARIABLE NAME>::=<LETTER>
         |<VARIABLE NAME><LETTER>
         |<VARIABLE NAME><DIGIT>
         |<VARIABLE NAME>'_'
         |<VARIABLE NAME>'''
                                                        2-3-1
<VARIABLE NAME LIST>::=<VARIABLE NAME>|
         <VARIABLE NAME LIST>','<VARIABLE NAME>
                                                        3
```

THIS STATEMENT IS NOT PART OF THE FORMAL DEFINITION, BUT IS INCLUDED FOR REFERENCE.

```
<KEYWORD>::=' ARITHMETIC'
         |'BLOCK '
         |' BY '
         |' CHARACTER'
         |' COLUMN'
         |'DEFINE '
         |' DIAGONAL'
         |' DO '
         |'ENDBLOCK'
         |'ENDIF'
         |'ENDFOR'
         |'FALSE'
         |'FINI'
         |'FOR '
         |'GO TO '
         |'IF '
         |' IN '
         |'LET '
         |' LOGICAL'
         |' LOWER TRIANGULAR'
         |'NULL'
         |' NONZEROS'
         |'OR IF '
         |'OTHERWISE '
         |'PROCEDURE '
         |' RECTANGULAR'
         |'RELEASE '
         |' ROW'
         |'SAME LOCATION '
         |' SET'
         |' SPARSE WITH '
         |' STEP '
         |' THEN '
         |'TRUE'
         |' UNTIL '
         |' UPPER TRIANGULAR'
         |' WHERE '
```

8 SAMPLE MPL PROGRAMS

```
PROCEDURE REVISED SIMPLEX(MATRIX,COSTS,RHS,BASIC_VARIABLES,
  UNBOUNDED,OBJECTIVE_VALUE,ITERATIONS)
DEFINE I,J:  "THESE ARE INDICES LATER ON"
UNBOUNDED := FALSE;  ITERATIONS := C;
LET P := MATRIX;
LET C := COSTS;
LET Q := RHS;
LET BV := BASIC_VARIABLES;
LET M := ROWDIM(P);
LET N := COLDIM(P);

          "WE ASSUME THAT BV CONSTITUTES A FEASIBLE SET
          OF BASIC VARIABLES GIVEN BY THEIR INDICES.
          WE WISH TO FIND X >= 0 SUCH THAT P*X = Q
          WHICH MINIMIZES C*X = OBJECTIVE_VALUE.  FIRST
          WE CALCULATE THE INVERSE OF THE BASES."
DEFINE INV_B M BY M;
INV_B:=INVERSE(P(*,BV));

          "THE CURRENT RIGHT HAND SIDE IS"
Q:=INV_B*Q;

          "THE CORRESPONDING COST VECTOR IS"
DEFINE CB M ROW;
CB:=C(BV);

          "S IS THE INDEX OF THE INCOMING COLUMN
           R IS THE INDEX OF THE OUTGOING COLUMN."
DEFINE S,R;

PRICING:BLOCK
ITERATIONS:=ITERATIONS+1;

          "FIND THE SIMPLEX MULTIPLIERS 'SM'"
DEFINE SM M ROW;
SM:=CB*INV_B;

          "AND THE SMALLEST RELATIVE COST FACTOR"
S:=ARGMIN(C-SM*P);

          "TEST FOR OPTIMALITY OF THE CURRENT BASIS"
IF C(S)>=SM*P(*,S) THEN
          "WE HAVE FOUND THE OPTIMAL BASIS"
  OBJECTIVE_VALUE:=CB*Q;
  RETURN;
ENDIF;
ENDBLOCK;

          "NOW COLUMN S IS INTRODUCED INTO THE BASIS.
          PB IS THE REPRESENTATION OF P(*,S) IN TERMS OF
          THE CURRENT BASIS"
DEFINE PB M COLUMN;
PB:= INV_B*P(*,S);
R:=0;
R:=ARGMIN(Q(I)/P(I,S) FOR I IN (1,....,M) | P(I,S)>0);
```

8 SAMPLE MPL PROGRAM (CONTINUED)

```
            "IF ALL P(I,S)<=0, THEN WE STILL HAVE R=0 AND
            A CLASS OF SOLUTIONS APPROACHING MINUS INFINITY
            EXISTS"
IF R=0 THEN
  UNBOUNDED := TRUE;
  RETURN;
ENDIF;

            "NOW UPDATE THE BASIC VARIABLE LIST BV, THE COST
            ASSOCIATED WITH THE BASIS L
            VECTOR CB ASSOCIATED WITH THE BASIS, THE VALUES
            Q OF THE BASIC VARIABLES, AND THE INVERSE
            INV_B OF THE BASIS."
BV(R):=S;
CB(R):=C(S);

            "UPDATE Q"
FOR J IN (1,...,M) | J¬=R, Q(J):= Q(J))-PB*(Q(R)/P(R,S));
Q(R):=Q(R)/PB(R,S);

            "NOW UPDATE THE BASIS INVERSE"
PIVOT(INV_B, PB,R);

            "NOW THE CYCLE IS COMPLETE AND WE RETURN TO
            CHECK THE OPTIMALITY OF THE NEW BASIS."
GO TO PRICING;
FINIS;

PROCEDURE PIVOT(MATRIX,PIVOT_COL,PIVOT,ROW)
LET M := MATRIX;
LET P := PIVOT_COL;
LET R := PIVOT_ROW;
FOR I IN ROWDOM(M) | I¬=R, M(I,*):=M(R,*)*(P(I)/P(R));
M(R,*):=M(R,*)/P(R);
RETURN;
FINIS;
```

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Computer Science Department Stanford University STANFORD, California 94305	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP
3. REPORT TITLE Mathematical Programming Language	
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Technical Report	
5. AUTHOR(S) (Last name, first name, initial) BAYER, Rudolf, BIGELOW, James H., DANTZIG, George B., GRIES, David J., McGRATH, Michael B., PINSKY, Paul D., SCHUCK, Stephen K., WITZGALL, Christoph.	
6. REPORT DATE May 15, 1968	7a. TOTAL NO. OF PAGES 91 7b. NO. OF REFS --
8a. CONTRACT OR GRANT NO. N-00014-67-A-0112-0011 b. PROJECT NO. NR-047-064 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
10. AVAILABILITY/LIMITATION NOTICES Distribution of this document is unlimited.	
11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Logistics and Mathematical Statistics Branch, Mathematical Sciences Division, Office of Naval Research, WASHINGTON, D.C. 20306

13. ABSTRACT

The purpose of MPL is to provide a language for writing mathematical programming algorithms that will be easier to write, to read, and to modify than those written in currently available computer languages. It is believed that the writing, testing, and modification of codes for solving large-scale linear programs will be a less formidable undertaking once MPL becomes available. It is hoped that by the Fall of 1968, work on a compiler for MPL will be well underway.

DD FORM 1 JAN 64 1473

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
Mathematical Programming						
Large-Scale Linear Programs						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year, or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through ________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through ________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through ________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as (TS), (S), (C), or (U).

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

DD FORM 1 JAN 64 1473 (BACK)

Unclassified
Security Classification